PhD-FSTM-2022-043
The Faculty of Science, Technology and Medicine

DISSERTATION

Defence held on 05/04/2022 in Esch-sur-Alzette

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Benjamin JAHIC
Born on 20 April 1991 in Tuzla (Bosnia and Herzegovina)

SEMKIS: A CONTRIBUTION TO SOFTWARE ENGINEERING METHODOLOGIES FOR NEURAL NETWORK DEVELOPMENT

I dedicate this dissertation to my beloved family, my parents Mirzeta & Zikret Jahić and my siblings Dženita & Alen Jahić, without whom this PhD thesis would not have been possible.

Thanks for their endless love, support and encouragement.

Declaration

I, Jahić Benjamin, declare that this dissertation entitled: "SEMKIS: A Contribution to Software Engineering Methodologies for Neural Network Development", and the work presented in it is my own. I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. Except such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Either none of this work has been published before submission, or parts of this work have been published as: [1–4]:

Jahić Benjamin
May 2022

Acknowledgements

I would like to express my sincere gratitude to my thesis supervisor Prof. Dr. Nicolas Guelfi and my thesis advisor Dr. Benoît Ries. Their enthusiasm, mental support, patience and suggestions made it possible to me to produce this research work in its present form. Their unique and brilliant supervision enriched this study higher than my expectation. This PhD thesis would not be possible without their support, inspiration and cooperation.

I would like to thank Prof. Dr. Holger Voos, Prof. Dr. Giovanna di Marzo and Prof. Dr. Leonardo Sousa for accepting to be part of my thesis jury and providing constructive feedback during the defense.

I would like to warmly thank my family, my beloved parents, Zikret and Mirzeta Jahić, who always support me, believe in me and whose words always motivate me to set and master many big challenges in life, my loving siblings, my sister Dženita and my brother Alen Jahić, who keep me grounded, remind me of what is important in life, and are always supportive of my adventures.

Thanks to my friends, who have supported me throughout my PhD studies. I will always appreciate everything they have done, especially Léon Berdé and Lisa Pulcinelli, who have always been by my side with advice and support; and have always motivated me to set and accomplish my goals!

Abstract

Today, there is a high demand for neural network-based software systems supporting humans during their daily activities. Neural networks are computer programs that simulate the behaviour of simplified human brains. These neural networks can be deployed on various devices e.g. cars, phones, medical devices...) in many domains (e.g. automotive industry, medicine...). To meet the high demand, software engineers require methods and tools to engineer these software systems for their customers.

Neural networks acquire their recognition skills e.g. recognising voice, image content...) from large datasets during a training process. Therefore, neural network engineering (NNE) shall not be only about designing and implementing neural network models, but also about dataset engineering (DSE). In the literature, there are no software engineering methodologies supporting DSE with precise dataset selection criteria for improving neural networks. Most traditional approaches focus only on improving the neural network's architecture or follow crafted approaches based on augmenting datasets with randomly gathered data. Moreover, they do not consider a comparative evaluation of the neural network's recognition skills and customer's requirements for building appropriate datasets.

In this thesis, we introduce a software engineering methodology (called SEMKIS) supported by a tool for engineering datasets with precise data selection criteria to improve neural networks. Our method considers mainly the improvement of neural networks through augmenting datasets with synthetic data. SEMKIS has been designed as a rigorous iterative process for guiding software engineers during their neural network-based projects.

The SEMKIS process is composed of many activities covering different development phases: requirements' specification; dataset and neural network engineering; recognition skills specification; dataset augmentation with synthetized data. We introduce the notion of key-properties, used all along the process in cooperation with a customer, to describe the recognition skills.

We define a domain-specific language (called SEMKIS-DSL) for the specification of the requirements and recognition skills. The SEMKIS-DSL grammar has been designed to support a comparative evaluation of the customer's requirements with the key-properties. We define a method for interpreting the specification and defining a dataset augmentation.

Lastly, we apply the SEMKIS process to a complete case study on the recognition of a meter counter. Our experiment shows a successful application of our process in a concrete example.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Abstract

This chapter starts with the presentation of the motivation and the context of this PhD thesis. We briefly describe the identified research problems in the domain of applying software engineering for engineering deep learning datasets. Afterwards, we summarise the main contributions of this thesis solving the described problems. Lastly, we present the document organization and provide a short overview of its content.

1.1 Motivation

Today's society increasingly requires a variety of Artificial Intelligence (AI)-based software systems supporting humans in their daily activities. There is an exponential demand for AI-based software systems in many domains (e.g. autonomous driving, finance, medicine...) AI-based software systems [14] are computer programs that simulate human intelligence performing a certain complex task. The use of such AI-based software systems has the potential to improve a human's daily activities, as for example recognising pedestrian on the road, recognising digits automatically in a meter counter states, identifying tumors in radiography...). A famous variety of AI-based software systems are the so-called neural networks[15], which simulate the behaviour of a simplified human brain model. An example of such, is the latest Google Translate application, which now uses a neural network[16] to translate languages. Over the last decades, neural networks have shown a successful capability in learning to perform the aforementioned complex task. Due to the humankinds' growing dependency on technological advances and the neural network's success, neural networks-based software systems are in growing demand nowadays.

In order to acquire the ability to perform a complex task, neural networks require large sets of data, called datasets. Neural network engineers are indeed expected to develop a neural network architecture and the required datasets for the training. They require methods and tool to efficiently and rapidly build high-quality neural networks together with the datasets. Therefore, dataset and neural network engineering activities are indispensable for the development of neural network-based software system. Traditionally, dataset and neural network engineering activities involve many error prone and time consuming human tasks.

Typically, neural network engineering activities are composed of the design, development and testing of the neural network. It involves different development steps such as the design of the neural network's architecture, specification of the architecture's parameters, training the neural network and testing the learnt recognition skills of the neural network.

Concerning dataset engineering activities, most datasets are typically hand-crafted without precise data selection criteria. Dataset engineering approaches mostly focus on randomly collecting and labelling data in their problem domain for constructing the datasets.

The AI community can benefit [17, 18] from the Software Engineering (SE) principles to escape from these traditional ad-hoc approaches to avoid inaccurate datasets and neural networks. SE appeared in the early 70s to solve the problem of engineering software with systematic and structured approaches. Since then, the SE research community has grown and helped to improve the engineering of software applications in various domains. SE helps in reducing costs, errors, failures and their impact by introducing rigorously specified processes and methods for software application development. Moreover, it is nowadays indispensable for the engineering of reliable and trustworthy software.

1.2 Problem statement

Traditional approaches do not follow a precise process and simplistic process, and are often error-prone. There are some efforts [19, 20, 5, 10] being made on defining software engineering approaches focusing on neural network engineering, but more rarely dataset engineering activities.

Concerning the current approaches on dataset engineering, the resulting datasets are often error-prone and lead to inaccurate neural networks after the training. These traditional dataset engineering approaches do not consider sufficiently the requirements, which leads to inconsistent and incomplete datasets. Neither do AI engineers define precise data selection criteria based on neural network's requirements to build appropriate datasets. Therefore, the resulting neural networks are often not satisfying the desired requirements. Either they are not able to recognise sufficiently the data, or they recognise more data then expected due to the broad spectrum of data. These inappropriate datasets lead often to high development costs and time consumption, because the datasets are mostly redesigned with randomly selected data leading to many neural network's training iterations. Moreover, collecting and labelling data is time-consuming and it produces high costs as it requires many ressources (e.g. man power, storage…) and time to retrieve the data. The costs are becoming even higher if multiple iterations for reconstructing the datasets and retraining the neural network are required.

Software engineers face these problems to build appropriate dataset for engineering neural networks for their customer's. Due to the lack of methods for precise data selection criteria based on customer's requirements, software engineers have difficulties for constructing optimal dataset for their neural network projects.

In summary, software engineers require methods for specifying the neural network's requirements, analysing trained neural networks and engineering datasets based on precise data selection criteria. In this thesis, we state the following research question that we would like to answer:

- *What is a software engineering methodology for constructing datasets to improve intelligent systems?*

1.3 Contribution

This thesis presents a novel methodology, called SEMKIS, standing for *Software engineering methodology for the knowledge management of intelligent systems*. It is important to mention that we defined SEMKIS as a long-term research project working on a methodology for the knowledge management of any kinds of intelligent system. In this thesis, we rather contribute to the domain with a first attempt leading towards a definition of SEMKIS. We consider intelligent systems as neural networks and knowledge as the neural network's recognition skills. In the upcoming chapters of this thesis, SEMKIS is considered as a methodology for engineering datasets in the context of improving the recognition skills of neural networks. Our main contributions are listed below.

First, we list our contributions related to our software engineering methodology for improving neural network with reengineered datasets. In this thesis, we introduce:

- An **iterative business process** for augmenting datasets to improve the neural network's recognition skills. We present the required activities and data objects in the context of neural network and dataset engineering. Our process covers different software engineering phases such as requirements engineering, design, implementation and testing. The main contribution of this process is the systematic augmentation of datasets with synthetic data based on precise data selection criteria for improving the neural network's recognition skills. We introduce the usage of a similarity function for the evaluation of the generated synthetic data. Finally, we present the different stakeholders exercising the presented activities.

- The notion of **neural network's key-properties** used for describing the neural network's learnt recognition skills during its training. The main contribution of this notion is to systematically specify the neural network's key-properties for defining the required datasets to improving neural networks until they satisfy the requirements.

- A **business process** for the specification and analysis of the neural network's key-properties. We present the required SEMKIS activities focusing on the specification of the neural network's key-properties based on the computed and observed information of the neural network's test results. The process covers different phase such as specification, evaluation and validation of the key-properties.

- A model driven engineering approach consisting of:
  - A **metamodel** defining the concepts related to the **neural network's requirements and key-properties**.
  - A **domain-specific language**, called SEMKIS-DSL, supporting the specification of the neural network's requirements and key-properties.

Second, we list our contributions related to our SEMKIS toolkit to support our methodology. In this thesis, we introduce the SEMKIS toolkit :

- A textual editor for specifying requirements and key-properties with the SEMKIS-DSL. The textual editor supports many features such as syntax highlighting, auto-completion, suggestions, templates...

- Technologies used to develop the toolkis: Xtext, Eclipse, Eclipse Modeling Framework, Xtend

The previously mentioned contributions have been experimented and validated on these two case studies:

- The MNIST case study: Recognition of handwritten digits.

- The counter meter case study: Recognition of the state of a digital synthesized meter counter.

1.4 Document organization

Chapter 2 introduces the required background concepts related to software engineering, deep learning, model-driven engineering and software engineering methodologies. It presents the state of the art related to our SEMKIS methodology and the added values of our work with respect to the literature.

Chapter 3 introduces our new software engineering methodology, called SEMKIS, designed as a rigorous process using the BPMN 2.0 modeling language [21]. It describes our new concepts, the different activities and data types of our process. Moreover, it illustrates the concepts of the SEMKIS process with the MNIST running example to provide a better understanding of our work.

Chapter 4 introduces the SEMKIS-DSL, a domain-specific language for the specification of the neural network's requirements and key-properties. It describes the conceptual model and the syntax with the grammar rules of the SEMKIS-DSL. Additionally, it introduces a method for interpreting the neural network's key-properties and generating a dataset augmentation specification. It described a method for performing a model transformation from the key-properties specification to a dataset augmentation specification. Finally, our tool support is describe in which we write the requirements and key-properties specifications using the SEMKIS-DSL.

Chapter 5 presents a concrete application of the SEMKIS methodology within a case study. The case study handles the problem of recognising the state of a counter meter (e.g. water counter meter, electricity counter meter...). We introduced the case study for experimenting our methodology in order to engineer (and reengineer) datasets to verify whether we are able to improve our neural networks.

Chapter 6 presents improvements and potential extensions of our SEMKIS methodology. We present few research tracks that can be investigated based on SEMKIS, as presented in thesis.

Chapter 7 presents a summary of our SEMKIS methodology, including our process, the domain-specific language and our main results; as well as its current limitations.

Finally, we appended some additional material to provide supplementary information to this thesis. **Appendix A** shows the grammar of the SEMKIS-DSL implemented using Xtext in Eclipse. **Appendix B** shows the concrete specification of the requirements and key-properties with the SEMKIS-DSL from our case study.

Chapter 2

Background & Related Work

Abstract

This chapter presents the background and related work of our SEMKIS methodology. We divided this chapter into two main sections presenting the background and related work concerning the SEMKIS dataset augmentation process and the SEMKIS domain-specific languages. For both sections, we start by briefly introducing and defining various general concepts used to produce our contributions as well as existing related work to these concepts. Afterwards, we present different papers from the literature. Concerning the SEMKIS process, we present many studies covering the usage of Software engineering (SE) principles in deep learning and other SE approaches to engineering datasets respectively neural networks. Concerning the SEMKIS-DSL, we present many studies related to requirements engineering (RE) for deep learning and many model driven engineering approaches using domain-specific languages to support the engineering of neural network and datasets.

2.1 The SEMKIS dataset augmentation process

In this section, we present the background and the related work concerning software engineering processes for building neural networks and deep learning datasets. We present the fundamental concepts required for understanding the SEMKIS process. Lastly, we present different processes, approaches and techniques that support neural network and dataset engineering.

2.1.1 Background

2.1.1.1 Artifical Intelligence, Deep Learning and Neural network engineering

In this subsection, we describe the most important notions in the fields of artificial intelligence and deep learning to understand the SEMKIS methodology and its process. In the context of deep learning, we describe the notions of neural networks as well as their training and testing. Finally, we present several technologies used for engineering neural networks.

What is artificial intelligence? Since decades, researchers and engineers have been working on artificial intelligence (AI) for supporting humans in their daily life. During this period, researcher have been trying to define the notion of artificial intelligence. Russel and Norvig [14] collected various definitions and summarised them into the following table:

Table 2.1 Some AI definitions categorised by Russel and Norvig in [14]

"The exciting new effort to make computers thin... machines with minds, in the full and literal sense" [22] *"The automation of activities that we associate with human thinking, activities such as decision-making, problem solving, learning... "* [24]	*"The study of mental faculties through the use of computational models"* [23] *"The study of the computations that make it possible to perceive, reason, and act"* [25]
"The art of creating machines that perform functions that require intelligence when performed by people" [26] *"The study of how to make computers do things at which, at the moment, people are better"* [28]	*"A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes"* [27] *"The branch of computer science that is concerned with the automation of intelligent behavior"* [29]

According to their work, the existing AI definitions can be grouped into four categories: human thinking, rational thinking, human acting, rational acting. Moreover, Russel and Norvig describe several approaches for verifying whether a system fits into these categories. From their work, we can conlude that defining artificial intelligence is a very complicated task. Nevertheless, it is possible to define an AI as, *software systems that think like humans, think rationally, act like humans and act rationally*. It is clear that more research must be investigated to create a system that fits into these four categories. Therefore, they propose different approaches to verify whether software systems fits into these categories.

Lastly, there exists various subfields of artificial intelligence targetting the development of various AI system types such as logical agents based systems (LA), machine learning systems (ML), deep learning systems(DL), robotics(ROB) . . . . In these domains, researchers and engineers are working on AI software concepts and models usable for building an AI system such as propositional logic-based programs(LA), vector machines(ML), random forests(ML), k-nearest neighbors(ML), neural networks(DL). This thesis studies methods and tools for engineering artificial intelligence. More precisely, we study artificial neural network as used in the deep learning domain.

What is deep learning? The word *deep learning* is composed of two distinct words, *deep* and *learning*. To understand the notion of deep learning, we separately describe these terms.

Learning refers to the capability of learning a certain task. In this thesis, we focus on algorithms that are able to learn a task, which are called learning algorithms. According to Mitchel [30], learning algorithms are computer programs that are able to learn and improve in performing some task (e.g. detecting tumors in x-ray images, detecting pedestrians in autonomous vehicles, translating a text. . .) from any data. The main differences [31] in between learning algorithms and traditional algorithms are :

1. A learning algorithm is not implemented explicitly with instructions that perform a targeted task.
2. A learning algorithm processes iteratively large datasets and updates internal parameters to learn performing a targeted task.

Deep refers to the architecture of the learning algorithm instances used in deep learning, which are trained on data to learn a targeted task. In deep learning, learning algorithms are typically designed as a connected and directed graph. These graphs are typically composed of multiple modules, which extract specific properties (called features or representations[32] in machine learning) from the processed data. The term *Deep* refers precisely to the number of modules used in the learning algorithm instance (called the depth). For example, these modules can be ordered sequentially, such that each module can use the computed properties of the previous module during the data processing. Thus, the higher the depth of this graph is, the more properties can be extracted by the learning algorithm.

Deep learning focuses on engineering learning algorithms, designed as connected directed graphs, which consist of a set of connected modules for extracting properties from data. An instance of these connected directed graphs are feed-forward artificial neural networks[33]. In the context of this thesis, we study aspects and approaches for improving neural networks.

What are neural networks? An important aspect in deep learning is the study of engineering neural networks. Neural networks [15, 34] are learning algorithms, which have been inspired from biology and designed as a simplified model of the human brain[35]. Neural networks are able to learn a desired task during a training process on a dataset. Researchers and engineers are working on different neural network architectures and methods for training the neural network to perform a desired task. Its architecture is designed as a connected directed graph consisting of sequentially connected modules. In deep learning terms, these modules are typically called layers. There exists different types of layers (e.g. input, output, hidden, convolutional, dropout, pooling, recurrent...) having different tasks during the processing of the data. For example, feed-forward neural networks are composed of an input layer, multiple hidden layers and an output layer. These layers consist of multiple nodes, called neurons, which are fully connected to the preceding and succeeding layers (except the input and output layers). Finally, each node is associated to some activation function[36] and initialized with some weight. During the training process, the neural network processes data through the entire network of nodes and updates the weights with an optimization algorithm[37]. The weights are updated until the neural networks reach a satisfying accuracy in performing the desired task. In this thesis, we study methods and tools for improving neural networks through analysing trained neural networks by reengineering the used dataset for training. In our case study, we experiement our methods with convolutional neural networks applied to different scenarios.

What are learning paradigms in the context of neural network training and testing? Neural networks have to be trained on datasets containing various different data elements. These data elements might be images, videos, voice messages, temperatures, pressures or any other kind of information. Depending on the targeted neural network's task and the dataset's design, it is possible to use different learning paradigms for training the neural networks. There are typically three types of learning paradigms used in deep learning: supervised learning[38], unsupervised learning[39] and reinforcement learning [40]. Since we emphasize more on supervised learning in this thesis, we will provide more attention to this learning paradigm. Cunningham[38] describes, in his work, supervised learning approaches. The aim of supervised learning is to direct the training of a neural network with the datasets in a desired way. Therefore, datasets are typically composed of labeled data. During the training, these labels are used to compute the error of the neural network's output (compared to the labels) for updating the weights with mathematical operations. Once the neural network's accuracy has reached a satisfying level, the training is stopped and the neural network is tested on a testing dataset. These testing datasets consist of various labeled data similar to the training dataset. However, both datasets are unique and do not share identical elements. The aim of the testing dataset is to verify, whether the neural network has learned the desired task and is able to recognise a sufficient amount of data. In this thesis, we designed and tested our methodology mainly for supervised learning problems. In contrary to the traditional neural network training methods, we do not focus on optimising the neural network's architecture to improve the accuracy. Our methodology focuses on iteratively reengineering the training dataset and retraining neural networks until we reach a satisfying neural network's accuracy. Thus, the neural network's weights are iteratively updated until the neural network recognises the expected output. Note that our methodology allows the optimisation of the neural network's architecture, which is out-of-scope of this thesis and left as future work. Nevertheless, our goal is to provide a sufficiently generalized method to provide a base for a software engineering methodology for unsupervised or reinforcement problems. Our methodology shall provide future opportunities for contributing to those two learning paradigms.

What are the most common technologies for engineering neural networks? There exists various frameworks and libraries supporting neural network engineering. Mostly, these frameworks have been developed for engineering neural networks with a specific programming language like Java[41], Python[42] or Swift[43]. For example, Keras[44] is an open-source library for the design of neural network's architectures. Other frameworks provide also backend-support allowing the execution, training and testing of neural networks. Example of such frameworks are pytorch[45], tensorflow[46], caffee[47], deeplearning4j[48], microsoft cognitive toolkit[49] and theano[50]... In this thesis, we use mainly Keras on top of tensorflow to design our neural network's architecture and execute our training/testing cycles. Additionally, we use many libraries (e.g. numpy[51], matplotlib[52] and bokeh[53]) for visualising the training progress as well as training/ testing results of neural networks.

2.1.1.2 Dataset engineering

In this section, we describe the most important notions related to dataset engineering in the context of deep learning. These notions are fundamental for the understanding of the SEMKIS methodology.

What is a dataset? Data[54] are any kind of collected information represented as numbers, text, images, videos or any other multimedia. Mostly data can be analysing for performing any kind of decision-making. In deep learning, a dataset[55] is a collection of data used for training and testing neural networks. The design of datasets may vary depending on the learning paradigm. In unsupervised learning, datasets contain only collected, or synthesized, data for training neural networks to recognise data clusters. In supervised learning, datasets typically consist of labeled data for training neural networks to recognise a label for the processed data. In reinforcement learning, datasets contain mainly information about the change of the environment states for training the neural network to operate in (or interact with) the environment. This thesis considers three types of datasets, training, testing and development dataset used during the engineering of neural networks. Since, we focus mainly on supervised learning problems, our datasets consist of labelled data.

What is dataset engineering? We use the notion of dataset engineering to describe the process of specifying, collecting, synthesising and analysing data to build appropriate datasets. The process involves different tasks such as the definition of the needed requirements, design of the dataset and construction of the dataset. In the context of our SEMKIS methodology, we present various dataset engineering activities focusing on the improvement of neural networks. Our method fundamentally relies on improving neural networks through engineering appropriate datasets.

What is meant by dataset augmentation? Dataset augmentation [56, 57] is the process of collecting, modifying and synthesising data and adding them to an existing dataset. It involves the development of methods and techniques for performing such a dataset augmentation. Existing data can either be collected from the real-world (e.g. photos, videos, x-ray scans...) or synthesized using some algorithm (e.g. virtual objects, game scenes, digitally modified images...). Depending on the data types, there exist various techniques for modifying datasets. For example, these are some existing techniques in the domain of image recognition for manipulating images to augment datasets consisting of images:

- image mirroring, crop, zoom, shift, rotation
- color space changes
- and many more

In deep learning, the aim of such dataset augmentation techniques is to increase the size of datasets to improve the training and testing of neural networks. In this thesis, we rely on dataset augmentation approaches to reengineer our datasets in order to improve neural networks. In our case study, we use similar image manipulation techniques to augment our datasets for retraining neural networks. Note that we use image manipulation techniques to illustrate our concepts of the SEMKIS methodologies. SEMKIS can also be applies to recognition problems (e.g. voice recognition, text recognition...), but we suggest to use other augmentation techniques appropriate to the problem domain. Dataset augmentation is one of the key concepts used for the design of our SEMKIS methodology.

2.1.1.3 Software engineering

What is software engineering? Sommerville [58] defines software engineering as an *engineering discipline* covering all development phases of software manufacturing. The development phases cover different engineering activities starting from requirements engineering until the production and maintenance of software systems.

Software engineering is about the development of theories, methods and tools for the production of software products. Software engineers require methods and tools to build their software products within their timing and budget constraints. To do so, it is required to develop a systematic, structured and well-ordered approach supporting engineers during their work. According to Sommerville, this is the most effective way to engineer advanced and well-functioning software. Additionally, he presents that software engineering helps to reduce engineering costs (at long term) as well as building reliable and trustworthy software.

In our context of neural network-based software systems engineering, traditional dataset engineering causes often a **high effort and high costs** [59]. The main issues are related to manual data collection and labelling, since large datasets are required to train and test neural networks. Without systematic methods and tools, software engineers are required to collect (or build) manually data. In supervised learning, data labelling is an exhausting task, which requires a lot of man power, ‘labelling’ volunteers and large software systems. The increasing size of humans required for bulding manually these dataset necessitates also large software and hardware systems.

In our context of safety-related systems, neural network’s customer want **reliable and trustworthy** software. In contrary to traditional software, neural networks are considered as probabilistic software that recognise the input data at a certain probability of certitude. It is therefore important to understand the neural network’s knowledge to avoid severe consequence. For example, possible failures and faults produced by neural networks used autonomous driving might be life-threatening.

What is a methodology, a method and an approach? A **software engineering methodology** is a framework that includes methods, procedures and tools to support software engineers to organise, execute and manage a development process of software products. Software engineering methodologies serve typically to improve the productivity and efficiency of the software production; as well to optimise the costs for delivering the final software product. A **software engineering method** is a precisely defined procedure including actors, steps and artefacts required for the development of software products. Methods can be designed as a business process, serving for guiding engineers during their software projects. In our context, a **business process** is an ordered set of activities performed by stakeholders to deliver a software product for a customer. The aim of a business process is to increase the customer satisfaction and optimise the engineering of software products. In this thesis, we use the concepts of business process to design our procedure for engineering datasets for neural networks. We use the BPMN 2.0[21] process modeling language for designing our business process. A **software engineering approach** is a set of ideas and assumptions about the way to deal with the development of software products. In our context, we use the term 'approach' to describe different engineering attempts, which are not supported by a precisely defined method (or for which no method exists).

What is a software engineering process model? The international standard, ISO 12207 [60], is a common framework for setting up software processes, denoted Software life cycle processes. They define a process as a *"set of interrelated or interacting activities that transforms inputs into outputs"*. This standard introduces a well-defined terminology for defining software processes. A life cycle model contains well-defined process' activities and tasks starting from requirements engineering until the termination of the use of a software product, including its development, operation and maintenace. Almost all existing software engineering process models define the activities and tasks of the following development phases:

1. Requirements specification phase
2. Design phase
3. Implementation phase
4. Testing and/or validation phase
5. Production and/or maintenance phase

In this thesis, we refer to a process model to define our activities and tasks required for engineering datasets for improving neural networks (our software product). We study methods and tools contributing mainly to the requirements specification and testing/validation phases. Several parts of our work contribute lightly to the design and implementation phase without formally defining methods or tools. We do not include activities and tasks related to the production and/or maintenance, neither do we study the engineering of neural network architectures.

What is meant by Software engineering for Deep learning? In this section, we briefly synthesize the three above introduced domain; deep learning, dataset engineering and software engineering. One key-aspect of *software engineering for deep learning* is about defining methodologies, including processes and tools, to support software engineers for building neural network-based software products. We have identified two different approaches that can be followed for improving neural networks:

1. Improving neural networks by reengineering its architecture. This approach involves updates of the architecture's parameters (weights, activation function, error function), changing the architecture type or inventing novel architectures.

2. Improving the training and testing datasets. This approach focuses mainly on understanding learned skills of a trained neural network for proposing an update of the datasets and retrain the neural network.

The first approach is mostly followed in the domain of machine and deep learning. Many researchers[61–67] study different neural network architectures to improve the learning process. In this thesis, we mainly study the second approach including the possibility to update also the neural network architecture.

In the context of software engineering, we study methods and tools for analysing trained neural networks and engineering improved datasets to support software engineers to build neural networks for their customers.

2.1.2 Related Work

In this section, we presented the related work to the application of software engineering to the deep learning domain. We present processes and approaches for engineering neural networks and datasets that are related to our SEMKIS process.

2.1.2.1 Software engineering for deep learning

In this section, we summarise 4 papers presenting some work related to software engineering (SE) applied to the domain of deep learning (DL). Since most related work focuses on a much larger spectrum, namely artificial intelligence(AI) and machine learning (ML), which is related to DL, We present some papers covering SE applied to these two domains (AI and ML).

Arpteg et al. [19] analysed different development approaches of seven DL projects to identify the main challenges regarding the application of SE to develop DL software. We summarise few of the identified SE challenges that are related to the :

- Implementation. (e.g. development libraries, hardware configurations, neural network model selection and configuration, dataset management. . .)
- Maintenance. (e.g. versioning of neural network models, data management, trustworthy development libraries)
- High data dependency for training and testing neural networks.
- Security. (e.g. neural network vulnerabilities, data safety. . .)
- Estimation of costs and time.

The authors conclude that it is still required to investigate further research to the development of DL software. According to them, the SE and the DL community shall collaborate together to improve the DL development processes to build high-quality tools and DL software. Menzies [18] analysed in his paper the importance of applying SE to develop AI. He introduces 5 rules to motivate the application of SE to AI. For each rule, the author presents particular examples from the literature motivating the usage of SE in the following contexts:

- Feasibility of applying SE to develop AI.
- Deployment and maintenance of AI software.
- Collection and design of datasets for producing AI.
- Application of SE methods for developing AI.
- Usability of AI in particular domain problems.

Khomh et al. [68] presents some particular problems in ML development that could be solved using SE principles. The authors present the following categories of problems related to different contexts :

- Development: building accurate software, data management, determining the usability of AI.

- Testing: complexity of testing AI, assuring the quality of AI software
- Maintenance: ensure that the models performs over time, reproducing a training process

The authors think that the SE and ML community should collaborate to solve these problems. Thus, they think that the communities could define engineering lifecycles that describe precisely the different development phases (e.g. design, implementation, testing, maintenance. . .) as well as processes that describe roles, artifacts and activities to solve the problems.

All authors of the above papers share a common idea: SE can help to develop AI, ML or DL based software systems. They contribute to the domain of applying software engineering to build AI software systems. Moreover, they identified data management as one of many AI engineering challenges for software engineers. In this thesis, we apply SE principles to improve the engineering of datasets for neural network based software systems. We define processes covering the different phases of the SE lifecycle for supporting dataset engineering to improve neural networks. We put a major focus on development phases such as requirements engineering, the design and the testing phase in the dataset engineering context.

2.1.2.2 Software engineering approaches for neural network development

In this section, we present two papers related to software engineering (SE) methods for building neural network-based software systems. Additionally, we present some other work that covers more general SE methods, approaches and guidelines focusing on building machine learning-based software systems. Nevertheless, we consider these studies as important, because their results can be used in the context of neural networks.

Senyard, Dart and Sterling [69] present major problems in developing neural network-based software systems. They claim that it is difficult to apply SE techniques to neural network development, because the development process differs from traditional software. For example, testing neural networks differs fundamentally from the testing of traditional software as it is required large dataset to be tested on in order to judge its acquired recognition skills acquired during the training. The stated neural network development problems are related to

- **Testing:** verification of the neural network implementation,
- **Design verification:** validating the design of datasets and neural networks before implementation.

- **Process repeatability:** reproduce a successful development of a neural network.
- **Software configuration management:** how to manage and maintain different neural network and dataset versions as well as deploying neural networks on different machines.

They introduce a framework, called neural network process model (NNPM), which is an extended version of a capability maturity model (CMM)[70] and targets to solve partially some identified problems. CMMs are a 5-level quality model, used to guide software engineers to improve and optimise their software development processes. Their NNPM consists of 5 maturity levels (like CMMs), where each level characterises different aspects of the software development process to increase the process in quality and maturity. Their framework starts with the definition of the first level, which characterises processes that are mainly improvised and executed without any further preparation or structured approach. The second level characterises neural network development processes that are planned and managed based on previous development experiences. The third level characterises the organisation and communication of development teams during the neural network development process for improving the productivity of neural networks. The last defined level (fourth level) characterises processes for that rely on quantitative metrics for assessing the neural network's accuracy and the data. The fifth level has not been defined because the current state of neural network development is premature to define such a process. In our work, we also think that there is a need for neural network development processes to support software engineers during their projects. However, our work differs in the sense that we define a concrete and precise process consisting of different activities supporting software engineers to build appropriate datasets and improve neural networks. We define a structured and organised approach to build reliable and trustworthy neural networks based on dataset engineering. We do not consider the analysis and characterisation of different process types.

Senyard, Kazmierczak and Sterling[20] present some methods for neural network engineering. In their work, they have identified four major problems in the development of neural networks.

1. Building neural network: Lack of processes to engineer neural networks
2. Design problems: It is hard to create some confidence to the design of neural networks. It is hard to prototype, inspects or evaluate the design of a neural network. The main reason is the updates of the neural network architecture's parameters (e.g. weights) during the training process. How to evalute the neural network's design before the training, if it changes dynamically changes over time?

3. The verification and validation of neural networks: How do we trust and verify that a neural network satisfies the requirements?
4. The SE community has not investigated many efforts in developing processes supporting neural network development.

They claim that neural networks are typically engineered with trial and error approaches, which makes it difficult to repeat the development process to rebuild the neural network. In their paper, they present an approach for designing neural networks. Additionally, they present an iterative process including the required actors supporting neural network development covering the specification of the requirements, the neural network implementation and the evaluation of the neural network. The presented process focuses mainly on updating the initial requirements or the neural network architecture's parameters in order to improve the training process and the resulting neural network. In our work, we present an iterative process for engineering appropriate datasets to improve neural networks. We focus on understanding the recognition skills of a trained neural network to update the datasets and redo the training. To update the datasets, we use techniques such as the generation of synthetic data to modify training datasets and retrain the neural network. In our process, we also consider the update of the neural network's architecture, but only after the dataset updates have let to an unsuccessful training.

Santhanam, Farchi and Pankratius [5] present an engineering lifecycle for engineering a software system containing one deep learning component (e.g. neural network...). Figure 2.1 shows the presented lifecycle of the paper. The lifecycle consists of two intersecting circles, where each represents a set of ordered activities for engineering software systems and a deep learning component being part of this software system. The activities are performed in clockwise order for both cycles. The first circle, called Application Lifecycle, covers the different engineering activities for building software systems. It starts with the requirements engineering activity as generally defined in SE. Other activities follow such as the software design, human-machine interaction, testing, deployment, monitoring, maintenance... After the human-machine interaction activity, the application lifecycle intersects the deep learning model cycle. The intersection is the start for the activities related to engineering the DL model. These DL model lifecycle activities are requirements engineering, dataset preparation, deep learning model implementation/training/testing tests and deep learning model integration into software system. Afterwards we return to the intersection, where the process continues with the remaining activities of the application lifecycle. Our work differs in the sense that we do not describe the integration of a deep learning component into a large software system. In this thesis, we focus on building an improved neural network by training it on precisely engineered datasets. Thus, we work on processes consisting of many SE activities

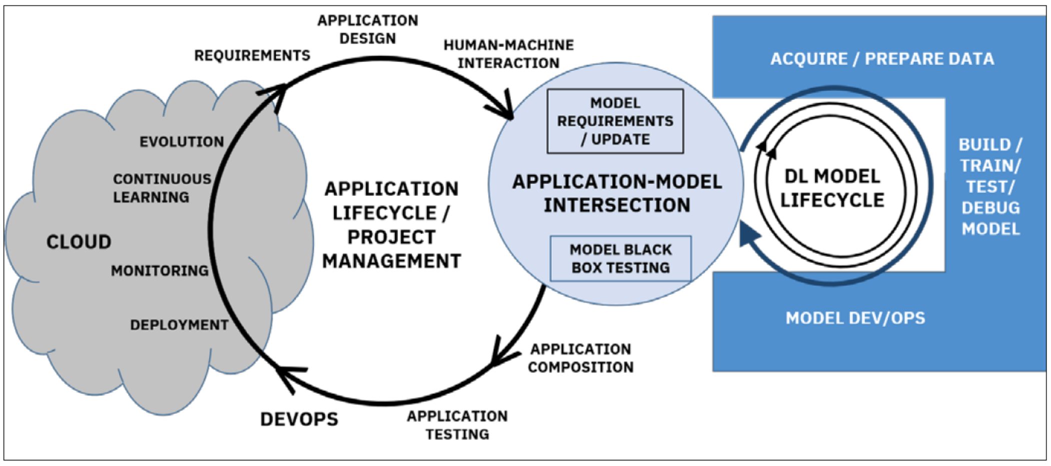


Fig. 2.1 Engineering lifecycle from Santhanam et al.[5].

defining precisely the tasks required for iteratively reengineering datasets to improve a neural network. It is possible to use the SEMKIS process to build an improved neural network that is integrable into a large software system. However, we do not define the required activities for performing the integration.

Amershi et al. [6] presented the workflow used at Microsoft for engineering machine learning (ML)-based software systems. They analysed the current development processes with surveys and interviews within the Microsoft engineering teams in order to define this general workflow. Their workflow consists of nine stages (e.g. model requirements, data management, feature engineering, implementation, testing, deployement and maintenance) for engineering the machine learning-based software systems. The presented Microsoft's workflow is shown in Figure 2.2, which contains the different stages. Additionally, they

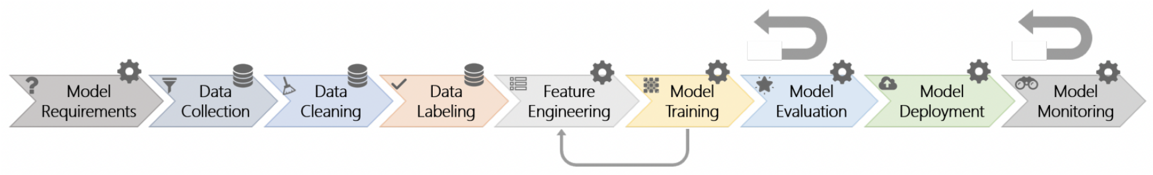


Fig. 2.2 Microsoft's workflow for engineering machine learning-based software systems from Amershi et al.[6].

present some best practices for developing ML-based software systems. We highlight the examples, which are the most related to our work:

- Data availability, quality and management: Data is required to engineer any ML software. Thus, it is required to have data for building appropriate datasets.

- Evaluation ML models: Systematic and structured approaches for evaluating ML models are important to validate a developed ML application.

Finally, they present a process maturity model consisting of 6 levels to support SE to improve their ML development processes at Microsoft. Each level of the maturity model defines some characteristics permitting to evaluate a ML development process. Our work differs in the sense that our process support dataset engineering for improving only neural networks. The authors follow mainly an approach to improve a ML architecture and they consider dataset engineering only in the beginning of their workflow. Our process uses dataset augmentation techniques to reengineer datasets with synthetic data for improving a ML architecture (in our case neural network architecture). We follow an approach of analysing a trained neural network and deducing a dataset augmentation, for reengineering datasets to retrain and improve a neural network. Additionally, we permit changes in the NN architecture, next to dataset changes, during each process iteration if we consider that neural network is not able to learn to recognise the data. However, the main focus of this thesis is not the improvement of a neural network's architecture, but focuses on reengineering dataset for improving neural networks. Moreover, feature engineering[71] is a less important activity, since neural networks are often trained directly on the datasets instead of extracted features from the dataset. Thus, our process considers the reengineering of the entire datasets instead of the improvement of the extracted features from the dataset.

John, Olsson and Bosch [7] studied the application of DevOps[72] concepts to engineering ML software. DevOps is about continuous software engineering principles[73] for defining methods and tools to produce/deploy efficiently and fastly high-quality software. In their work, they used the DevOps practices to improve the delivery of ML software. Therefore, they propose the framework illustrated in Figure 2.3. The framework consists of three pipelines (data, modelling and release pipeline). Each pipeline represents a set of automated processes and tools for these goals:

- Data pipeline: processes and tools for automatising the data management (collecting, labelling, preprocessing, feature engineering...).
- Model pipeline: processes and tools for automatising the execution, optimisation, evaluation of machine learning models.
- Release pipeline: processes and tools for automatising the deployment of the machine learning model.

The authors have analysed current ML development approaches to derive the framework from them. The framework is a high-level description of the concepts required for continuous ML

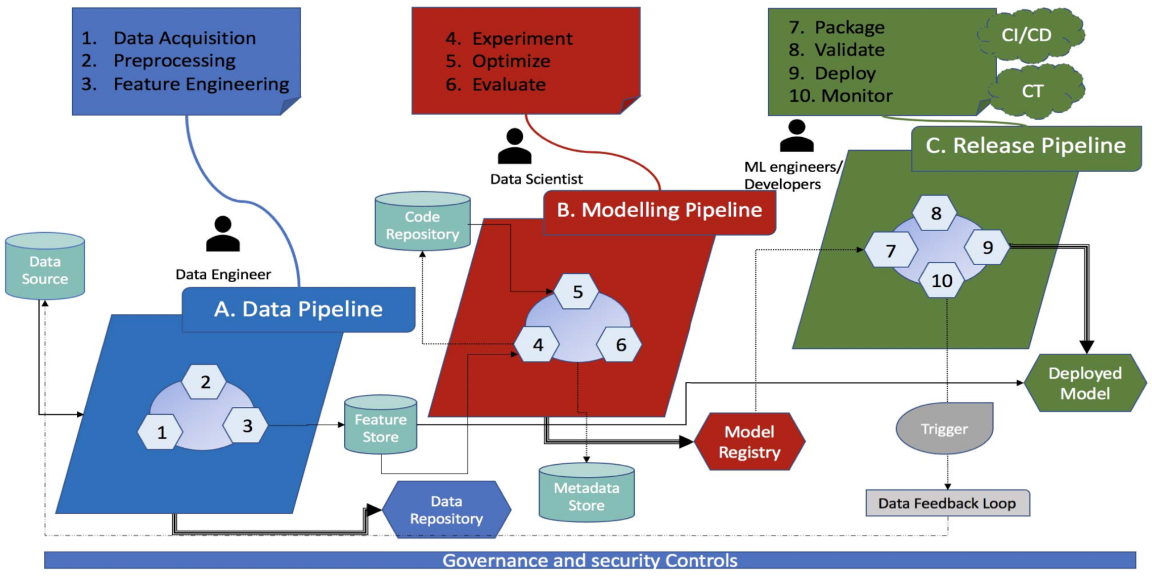


Fig. 2.3 MLOps Framework defined from John, Olsson and Bosch [7].

software development. In their paper, they do not precisely define the design and construction of the different pipelines. However, they present a maturity model consisting of different levels described different characteristics concerning the automatisation of development process (data collection, model deployment, semi- and fully automated model monitoring). They used the maturity model to evaluate three companies to make some propositions for improving the continuous ML software development. In our work, we introduce a process with precisely defined activities and tasks for building datasets to improve neural networks. We define precise tasks to support software engineering for developing improved neural networks. Moreover, we introduce a concrete tool supporting the specification of the neural network's key-properties and requirements.

Hamada et al. [74] present guidelines for evaluating the quality of ML-based software system. The aim of their work is to support engineers with a concrete approach for improving the quality of ML software by understanding their characteristics. To do so, they have introduced the notion of quality of ML software defined by some general criteria. The presented global criteria are data integrity, model robustness, system quality, process agility, customer expection. For each quality criteria, they defined multiple checkpoints that must be verifyed and validated in order to satisfy the criteria. Thus, engineers can verify whether the checkpoints are valid in order to assure the quality of the ML software. Moreover, they introduced some concrete approaches for verifying the quality of ML software in the context of some domain-specific problems such as visual recognition, autonomous driving, generative systems. . . Our work differs in the sense that we target in our process to improve neural networks by improving the satisfaction of the customer's requirements. In SEMKIS, we target the understanding of the neural network by its learned recognition skills. We use the recognition skills to verify if the requirements have been satisfied in order to validate our neural network.

Hesenius et al. [8] present a software engineering process, called EDDA, to support the development AI software and datasets. The authors present some typical problems (e.g. anomaly detection, clustering, regression, classification...) that can be solved using ML software. They claim that the development approach (in its core) for any ML software is the same for any of these problems. The authors argue also that the development of ML software is usually a subproject of a larger traditional software system. So, they propose a general process, called EDDA, which consists of two different subprocesses to support software engineers to develop ML-based software systems. Figure 2.4 illustrates the process, which is presented in the paper.

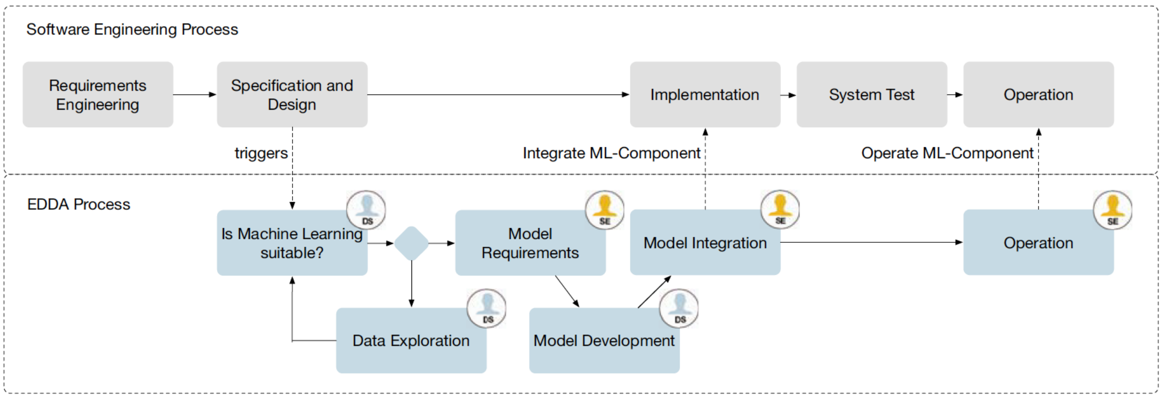


Fig. 2.4 EDDA software engineering process overview from Hesenius et al. [8].

The first subprocess contains the traditional software engineering phases from the SE development lifecycle (e.g. Requirements engineering, Design, Implementation, Testing and Maintenance). The second subprocess contains different activities related to ML model development. We summarise the process' activities in the following list:

1. Verify whether ML is suitable for a given problem
2. Explore the required data
3. Specify the requirements for the ML software
4. Implement the ML model
5. Integrate the ML model into the larger software system.

6. Maintain the ML model

The activities of the second subprocess are interconnected with the SE lifecycle phases of the first subprocess. Thus, the second process is triggered and started during the specification and design phase of the larger software system, where the software engineer is supposed to verify whether ML is suitable to solve their problem. It ends with the integration of the ML software into the larget software system and the maintenance of the ML model. The authors refined also different activities and describe the different tasks to execute the process. Another aspect is that the authors introduce four different actors (illustrated in Figure 2.5), namely the domain expert, data scientist, data domain expert and software engineer. These actors play a central role in the execution of the different activities. Each activity respectively task of the EDDA process is associated to an actor, who is responsible for its execution. The authors conclude that it is also required to investigate more research to refine the process with their increasing experiences in many ML development projects. Our work differs in multiple

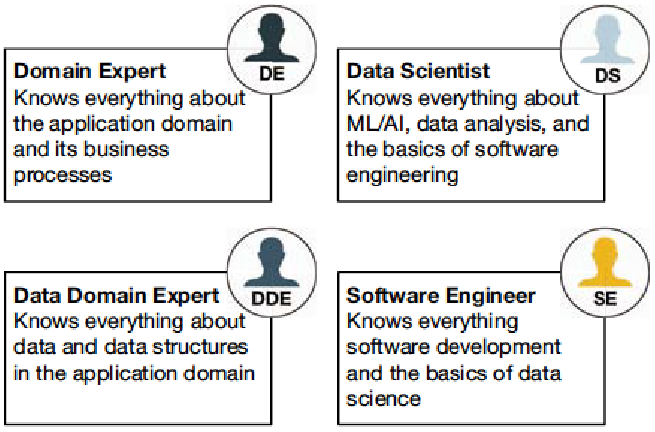

Domain Expert
Knows everything about the application domain and its business processes
DE
Data Scientist
Knows everything about ML/AI, data analysis, and the basics of software engineering
DS
Data Domain Expert
Knows everything about data and data structures in the application domain
DDE
Software Engineer
Knows everything software development and the basics of data science
SE

Fig. 2.5 EDDA actors overview from Hesenius et al. [8].

aspects. First, we propose a software engineering process defining precise activities to build datasets and engineer exclusively improved neural networks that satisfy the customer's requirements. Secondly, we do not propose a general process for any kind of AI/ML problem. We developed a process for building datasets for engineering neural networks that solve supervised learning problems. Thirdly, our method relies on real and synthetic data to construct our datasets. Fourthly, our fundamental goal of SEMKIS is to continuously update both, our datasets and our neural network architecture, to develop a neural network that satisfies the customer's requirements. In our process, the development of dataset plays a central role as we improve our neural networks by training them on synthetically augmented datasets.

2.1.2.3 Software engineering approaches for building datasets for neural network development

In this section, we present many state-of-the-art dataset engineering approaches. We found some approaches to build datasets for machine learning and deep learning models. Among others, we present some dataset engineering lifecycles, dataset engineering guidelines, different data modification and augmentation techniques...

Roh et al. [10] studied some state-of-the-art data collection approaches from the machine learning and data management community. The authors have categorised various techniques for collecting data in order to build datasets for training and testing ML software. The selection of a data collection technique depends on the domain problem and the dataset availability (e.g. existing dataset, labelled data, no datasets available...) Thus, the authors propose first contributions to improve the quality of datasets by optimising the selection of data acquisition techniques. In this article, they present these three data collection activities:

- Data acquisition - Focuses on finding existing datasets for training a DL software.
- Data labeling - Focuses on labeling data in existing and available datasets.
- Existing data improvement - Revising datasets by correcting labels and/or adding/removing data.

According to them, each activity can be performed with different techniques consisting of different tasks. For example in the context of data acquisition, they present that data can be acquired by using data discovery, data augmentation or data generation techniques. Figure 2.6 from the paper presents an overview of different data collection categories and techniques. Additionally, they define a decision flow chart for deciding which techniques shall be used to construct a dataset, label dataset for selecting a learning technique for the ML software, improve existing datasets by revising its data elements. If none of the techniques can be used to improve the accuracy of an ML software, then the ML software shall be revised. Figure 2.7 shows the decision flow chart presented in the paper.

Our method differs in the sense that we design datasets to build appropriate dataset for improving neural networks in the context of software engineering. In SEMKIS, we consider the customer's requirements to collect our data for building appropriate datasets for training and testing the neural network. The aim of our method is to define precise activities for software engineers to build appropriate dataset with the goal to improve the satisfaction of the neural network's requirements. We apply dataset augmentation and data revising techniques to reengineer our datasets after having analysed a trained neural network. While constructing

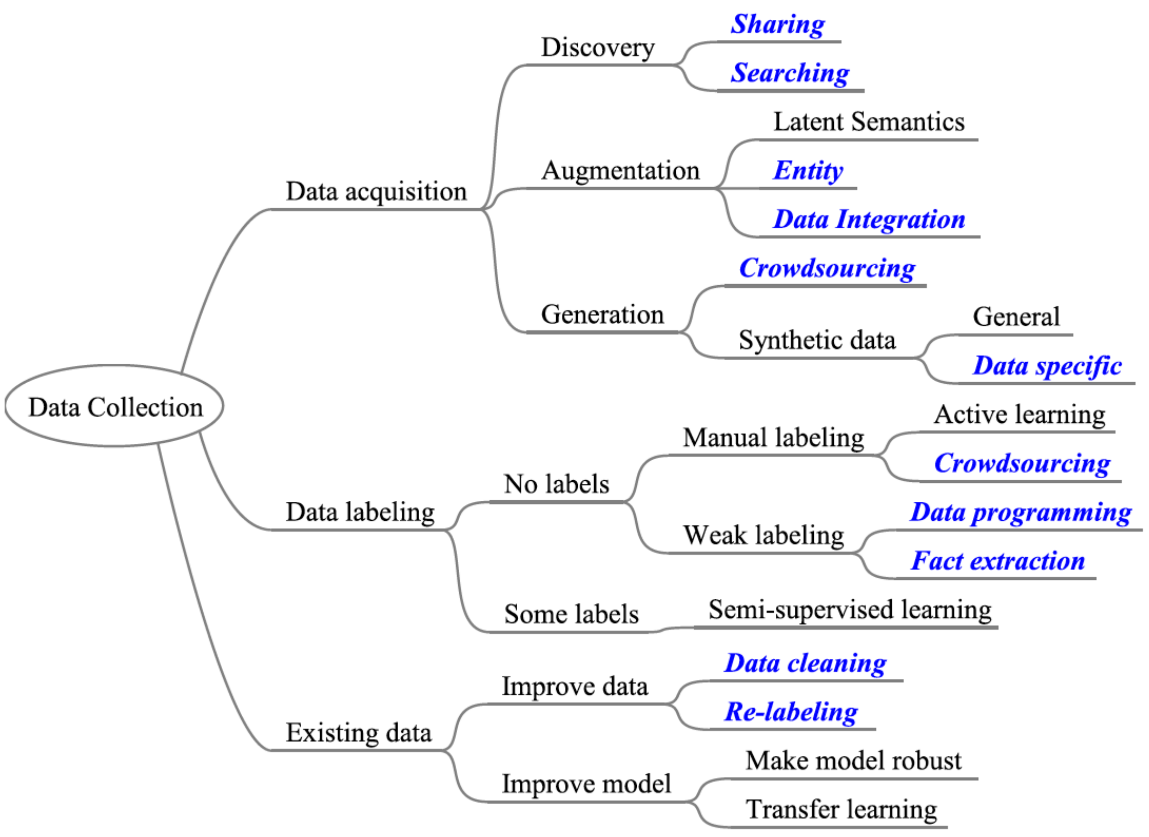


Fig. 2.6 Data collection approaches presented by Whand and Lee in [9].

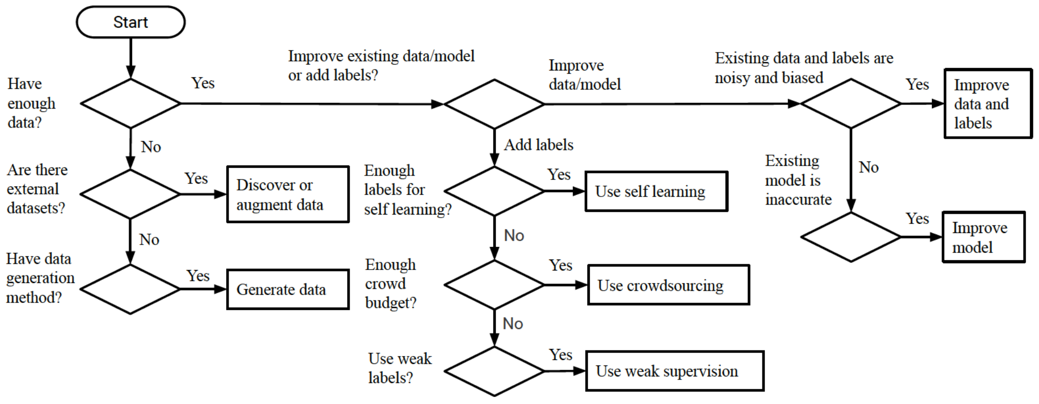


Fig. 2.7 Decision flowchart for selection data collection techniques to build dataset by Roh et al. [10].

the input data, the software engineer is supposed to collect or synthesize the required data based on the input requirements given by the customer.

Whang and Lee[9] present a tutorial for building dataset to train DL models. Their work has been designed based on the previously described paper from Roh et al.[10]. First, the authors present a global process for engineering neural networks. The process consists of 5 activities (including dataset and neural network engineering activities): data collection, data clearning and validation, model training, model evaluation, model management and serving. In the remaining part of the paper, the authors focus on presenting a tutorial describing how to perform the first 3 activities: data collection, data clearning and validation, model training. The authors do not define formally a precise process describing the tasks that shall be performed during three activities. They are introducing some guidelines and some best practices for performing the required tasks to acomplish certain activities. The aim of their tutorial is to guide engineers to build datasets that are capable of training and evaluating a DL model. Their goal is to improve the global accuracy of DL models with appropriate dataset. In this thesis, we also target to improve DL models (neural network in our context) with appropriate datasets. However, we consider the customer's requirements and the neural network's recognition skills to define precise data selection for engineering our appropriate datasets. We use these appropriate datasets to improve neural networks to satisfy the customer's requirements. Moreover, our process differs in the sense that we define precisely several activities and subprocesses related to the specification of the requriements, the engineering of datasets, the engineering of a neural network, the testing and analysis of a neural network... The process is designed to support software engineers during the DL development projects.

Hutchinson et al.[11] present a rigorous framework for the development of accountable and transparent datasets. The authors present a dataset development life cycle consisting of 5 phases inspired from the software engineering lifecycle. The 5 phases are the requirement's analysis, design, implementation, testing, maintenance. Each phase describe different characteristics related to dataset engineering. Figure 2.8 shows the presented lifecycle in their article. In the paper, the importance of a dataset development lifecycle is highlighted to reduce the number of failures related to data-issues and to improve ML software. The authors follow an approach to improve dataset engineering with precise documentation of the different phases of the lifecycle. For each phase of the development lifecycle, the authors have defined some questions, which have been answered by data engineers. The questions and answers help and guide the engineers to build appropriate datasets. Each phase defines different characteristics related to the selection of data and construction of the dataset by meeting the input requirements. Our work differs in the sense that we consider the neural network's recognitions skills to engineer our datasets. Since it is complicated to understand the neural network's learning, we consider also the analysis of a trained neural network in order to deduce some dataset changes. Thus, we iteratively verify the neural network's recognition skills in order to apply some dataset modifications (e.g. dataset augmentation).

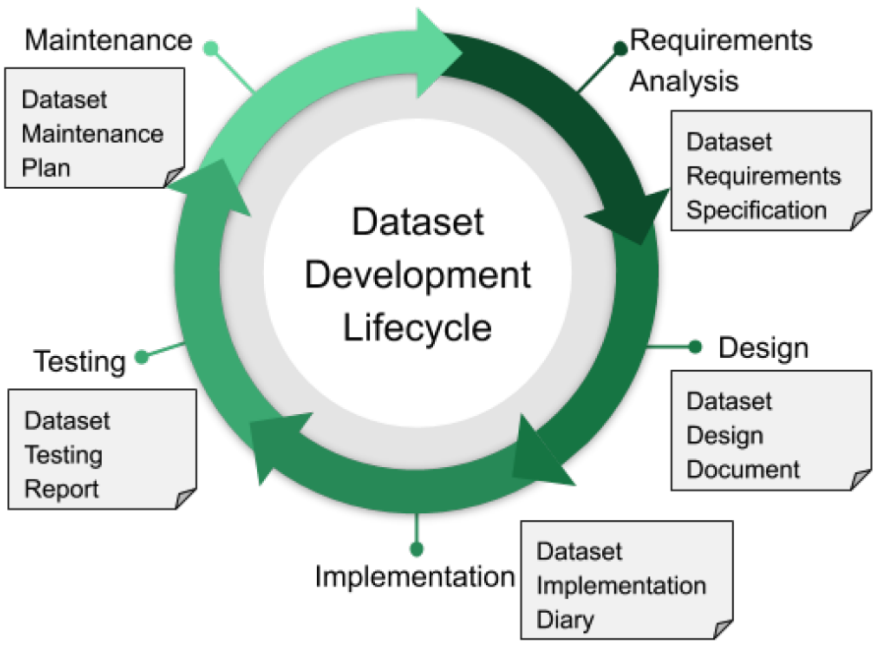


Fig. 2.8 Dataset Development Lifecycle presented by Hutchinson et al.[11].

In the remaining part of this section, we present some examples related to traditional dataset engineering approaches. Traditional dataset engineering is usually performed using ad-hoc approaches, which are often based on collecting randomly or synthesizing a large amount of data from the problem domain. These data are then used to train and test the neural network, which are able to learn the targeted skills. In the literature, we found many techniques to construct a dataset for training and testing neural networks (or other ML models).

A first widely used approach is to construct datasets with randomly collected data. Many datasets[75–78] have been constructed with randomly collected data from the problem domain. Engineers collect randomly a large amount of data (without precise data selection criteria or precisely specified requirements) in order to maximise the data coverage describing the problem domain. Then, the datasets are used to train a neural network to learn the desired skills (e.g. learn the labels of each data element in the dataset.). Many of these datasets are used to evaluate and compare different ML models. Secondly, some datasets have been constructed with synthesized data.

A second used approach is the collection of synthetic data gathered from a virtual world (e.g. games, 3D virtual models…) Many datasets[79–85] consists of such synthetic data obtained from a virtual world. This approach is often used in the problems of the image/object recognition domain. (e.g. autonomous driving, furniture detection, handwritten digit recognition. . .) Instead of collecting randomly real-world data, which is very costly and time-consuming, data is often synthesized and collected from real-world simulations. The collected data is then used to construct the dataset. Similar to the previous technique, these datasets are often constructed with randomly synthesized data to increase the dataset size.

A third used approach is to modify existing data in order to increase the dataset size for training and testing neural networks. There exists various techniques[86–93] supporting the modification of data (e.g. image cropping, flipping, rotating. . .). The synthesized images are then used to augment a dataset with additional data. Similar to the previous techniques, these datasets consist of a large amount of randomly generated data to increase the dataset size.

A fourth used approach is to generate automatically a large amount of data for increasing the dataset size. These approaches are often based on neural networks[94–96]. For examples, generative adversarial networks[97, 98] are used to generate randomly some data (e.g. images. . .). The generated images are then used to increase the dataset size. Other techniques[99, 100] target a fully-automatised approach for collecting or synthesing random data. For example, autoAugment[99] uses neural networks and reinforcement learning to find the best data augmentation policy. The aim is automatically find and perform the best data augmentation (using a neural network) for a specific problem in order to improve a neural network. These approaches and techniques differ from our work in the sense that we collect and synthesize data with precise data selection criteria (e.g. the customer's requirements). In our method, we decide which data is needed based on the learned recognition skills of a neural network. Thus, we precisely generate the required data to augment the datasets in order to improve the neural network such that it satisfies the customer's requirements.

2.2 The SEMKIS domain-specific language

In this section, we present the background and the related work concerning model-driven engineering approaches and domain-specific languages for improving neural network and dataset engineering. First, we present the fundamental global concepts related to Chapter 4 presenting the SEMKIS-DSL. Lastly, we present existing model-driven engineering approaches, that rely on domain-specific languages, and the application of requirements engineering in the domain of deep learning.

2.2.1 Background

Model-Driven Engineering

In this section, we describe the concepts related to model-driven engineering and important notions related to it. These concepts are required to understand our thesis' contributions related to the SEMKIS-DSL presented in Chapter 4.

What is a model-driven engineering? As described by Fondement and Silaghi [101], model-driven engineering(MDE) is a software engineering methodology relying on the specification of domain models describing together the domain problem and its solution. Domain models are conceptual specifications of a particular domain problem described at different level of abstractions. MDE applies a top-down approach, starting with the model design at the highest level of abstraction, which is then used to derive a refined lower abstraction level model. Each refinement process has to be precisely defined, in order to include additional conceptual details to each subsequent abstraction level integrates which are missing in higher abstraction level models. The aim of this approach is that each subsequent model is derived from its preceding model and specified with a lower level of abstraction. Moreover, the specified model can be used to define code generators or interpreters permitting to generate source code in a programming language. Once, the lowest abstraction level model is designed, it is possible to generate a source code, implementing the model, in a defined programming language. The greatest benefit of this approach is that, once a change has to be performed on a given model, the changes impact on the subsequent models can be quickly detected, and the models adapted to the change if required. Thus, a new source code can be generated with the updated models. In the following, we will describe four important notions often used in the context of model driven engineering.

In this thesis, we contribute to a model-driven engineering approach to automatise the generation of synthetic data for augmenting datasets based on requirements and key-properties specifications. We propose a domain-specific language for specifying the requirements and key-properties as well as a manual approach to interpret the specification and augment a dataset. The fully automated approach remains a future work option for further research.

What is a metamodel? Fondement F. & Silaghi R. [101] define a metamodel as a high level abstraction model defining the modeling language using concepts and relations. Furthermore, a metamodel defines the notations and abstract syntax to be followed for the design of the models at each abstraction level and can also include the definition of the semantics and the concrete syntax.

What is a model? As defined by Bézivin J. & Gerbé O. [102], a model is a simplification or refinement of the metamodel, in other words, it is derived from the metamodel. It is often easier to use and understand than the metamodel itself, since its abstraction level is always lower than the abstraction level of the metamodel.

What is an instance? An instance is a direct derivative of a model. To illustrate this, we can take a random metamodel and derive a model from it. The derived model is called an instance of the metamodel.

What is a model transformation? Khalil A. & Dingel J. [103] describe model transformation as the act of transforming a source model into a target model with some transformation rules. In general, there are two existing types of model transformations which can be applied:

- **Endogenous transformation** Transform a source model into a target model complying to the same metamodel
  - Used for model refinement or optimization
- **Exogenous transformation** Transform a source model into a target model complying to a different metamodel
  - Used for code generation

In this thesis, we designe a UML diagram, representing our metamodel and the concepts for specification of the requirements as well as the key-properties. We directly derived the SEMKIS-DSL grammar from the UML-diagram, which we consider as our model. We consider concrete specifications in the SEMKIS-DSL as our specification instances. Finally, we consider as model transformation the interpretation of a requirements and key-properties specification model into a dataset augmentation model.

Domain-specific languages

In this section, we describe domain-specific language and important notions related to it.

What is domain-specific language? Kosar et al.[104], describe a domain-specific language as a high abstraction level language, textual or graphical, which is tailor-made for a specific application domain and thus is based on the concepts and characteristics of that domain. In other words, a domain-specific language is not a general purpose language, usable for any kind of problem, but is only limited to the problems in the domain it was designed for.

What is the syntax of a language? The syntax of a language, is a set of defined rules the ordering of symbols, words, punctuations and statements of a language, which according to Fondement F.[105], can be broken down into two distinct types:

- **Abstract syntax** The abstract syntax is a high abstraction level syntax describing the concept and structure of the language. It is essentially used by compilers for the internal representation.
- **Concrete syntax** The concrete syntax defines how the language elements of the abstract syntax are used, thus provides a user-friendly way to write programs in the language.

In this thesis, we develop a domain-specific language, called SEMKIS-DSL, which is designed only for specifying the customer's requirements and neural network's key-properties to support the engineering of improved neural networks.

Technologies

Which tools can be used for modeling software? In order to apply model driven engineering to model software, one has to chose between a rich variety of tools available. In this thesis, we focus on two widely used modeling tools, namely, Eclipse Modeling Framework(EMF) and Unified Modeling Language (UML).

Eclipse Modeling Framework (EMF) EMF is the Eclipse based modeling framework [106] allowing the creation of models using specifications written in XML. Furthermore, it allows code generation in the java programming language, by generating the set of classes implementing the model. In this thesis, we use indirectly EMF through the Xtext Framework (introduced later in this section) for specifying our models.

Unified Modeling Language (UML) UML [107] is a visual, general purpose, modeling language offering a rich set of features allowing successful creation of any kind of model. It is widely used in the domain of software engineering as it allows software engineers to visualize complex software systems using models, which make the software system more understandable. In this thesis, we use UML to design our conceptual model for the requirements and key-properties specification.

Which tools can be used to design and implement domain-specific languages? The design and implementation of a domain-specific language requires the selection of a tool offering the required functionalities to do so, e.g. Xtext, Jetbrains MPS[108], etc. In this thesis, we use Xtext to design and implement the SEMKIS-DSL grammar. Xtext is an open-source framework for implementing textual domain-specific languages. It offers many features such as syntax highlighting, code templates, auto-completion, code generation, hover... Moreover, Xtext has been developed as a plug-in for the Eclipse IDE, which is a widely used platform with a large global community and many forums. The Xtext tool itself is well experimented and known with also a large community and active forums, which offers a good technical support. Our research team, the Messir team, has many years of experience in the engineering of domain-specific languages using the Xtext framework (e.g. Messir[109] and Tesma[110]).

What are the technologies to design and implement a model transformation? For the model transformation, in particular, for the transformation of an input model into usable code, one has to select a so-called model-to-text transformation which takes as input a model, created using a modeling software, and generates usable code in the required programming language. Hence, one has to select the code generator by taking into account the compatiblity with the modeling software and the required programming language. In this thesis, we use Xtext to design our SEMKIS-DSL. We have not implemented yet an automated code generator, but Xtext delivers code generator libraries, which are implemented using Xtend, a programming language for the Java Virtual Machine[111]. Thus, Xtext offers different features to implement a generator permitting to transform a SEMKIS specification model into other models (e.g. neural networks implemented in Python[42]). Concerning other tools, models designed using EMF can also be transformed into source code using the EMF code generator, which itself, has been modeled in EMF. UML models can be transformed into source code using available code generators, e.g. Visual-Paradigm[112], Software Ideas Modeler.

2.2.2 Related Work

In this section, we present the related work to the application of software engineering to the deep learning domain. We present processes and approaches for engineering neural networks and datasets that are related to our SEMKIS process.

2.2.2.1 Requirements engineering for deep learning

In this section, we present different contributions related to the application of requirements engineering (RE) to improve the development of deep learning (DL) software. We found very few papers related to RE approaches to support the engineering of deep learning software.

Since most contributions, that we found in the literature, is related to RE for developing artificial intelligence (AI) or machine learning (ML), we decided to include these papers as well. We present mainly the importance and ongoing research of RE in the fields of AI, ML and DL.

Villamizar, Escovedo and Kalinowski[113] present a systematic mapping study to identify different characteristics on RE for ML in the literature. The authors collected and analysed 35 studies published in conferences and journals in between 2018 and 2021. These collected studies present different contributions related to various RE activities in order to improve the development of ML software. These contributions are categorized in RE approaches, machine learning engineering checklists and guidelines, machine learning quality models and taxonomies categories. The authors identified that requirements elicitation and requirements analysis are the main RE activities considered by the studies. In this context, most papers focus on the prioritization of requirements as well as defining customer's expectations. Another aspect, which the authors present are contributions related to non-functional requirements for machine learning software. These contribtions define different quality properties such as data quality, safety, transparency, fairness. . . The authors found many approaches related to the development of ML software. Except of two papers (our SEMKIS approach[4] and Challe et al. [114]), all proposed approaches target to support the development of ML software for any kind of problem (e.g. supervised, unsupervised and reinforcement learning problems)

Due to the increasing amount of studies, the authors underline the importance of requirements engineering to support requirements engineering during their machine learning projects. Among others, they have identified the following challenges in RE for ML:

- Lack of validation techniques (e.g. specifying correctly requirements for machine learning, quality metrics. . .).
- Complexity and difficulty to understand nonfunctional requirements in machine learning.
- Satisfiability of the customer's requirements.
- Customer's limited comprehension of ML software.

Thus, research must investitage to offer approaches and methods to improve the communication with the customer and build a ML software that satisfies the customer's requirements. Moreover, the authors claim that most approaches presented in the papers are too optimistic by covering the development of machine learning software for supervised, unsupervised and reinforcemenent learning problems.

Many other papers [115–117] present similar challenges, problems and characteristics related to requirements engineering for machine learning. The authors of the papers agree that requirements engineering becomes increasingly important with the advances of AI software development. Thus, further research shall be investigated to propose methods and tools for requirement engineering for AI software The authors present many significant challenges related to the definition of quality attributes (e.g. reliability) of AI/ML software systems, the role of non-functional requirements(NFRs) for AI/ML software and approaches for requirements engineering to support AI software development. We summarize some presented challenges in the following list:

- Definition of non-functional requirements for AI/ML systems.
  - Definition of new NFRs for AI/ML systems such as accuracy, explainability, repeatability, flexibility, legal requirements (e.g. accident responsability), humanity (e.g. freedom from discrimination).
  - Verification of the reusability of NFRs for traditional software (e.g. testability, correctness, reliability, safety).
  - Definition of the trustworthiness of the skills learned by AI/ML software.
- Definition of requirements related to data.
  - Approaches for precise comprehension of the context/domain in which an AI/ML-system is applied in order to specify some requirements.
  - Definition and management of data quality requirements.
  - NFRs are often not considered for the selection of data, but play a major role for the AI/ML model itself.
- Definition of the AI/ML model performance.
  - Comprehension of the AI/ML performance to specify appropriate functional requirements.
  - Metrics to measure the AI models performance in a specific context.
- Specification, documentation and maintenance of requirements
  - Integration of ML specific requirements into a traditional requirement engineering process.
  - Methods and tools for the specification and maintenance of requirements for AI/ML software.

– Verification of the satisfiability of the requirements.

In this thesis, we apply software engineering techniques to design our SEMKIS methodology to support software engineers to develop datasets and neural networks. Therefore, we also consider requirement engineering as an important activity to develop improved datasets and neural networks. In SEMKIS, we define various concepts permitting to define functional and non-functional requirements for building datasets and neural networks. Our aim is to support software engineers to engineer approriate neural networks for their customers by understanding and specifying their customer's needs. In our method, we propose the specification of the requirements for engineering dataset and neural networks in the context of supervised learning problems. We target the comprehension of the customer's needs to specify precisely the requirements for developing the corresponding neural network. Moreover, we validate the requirments specification by comparing them with the acquired neural network's recognition skills after the training.

Rahimi et al. [12] present an approach to improve the specification of the requirements for machine learning software. The authors target to support requirements engineers with techniques and tools to analyse and engineer requirement specifications for machine learning software. Their aim is to specify complex concepts to improve the construction of datasets and the corresponding ML model. They follow an approach that is based on precise specifications of the problem domain in order to identify weaknesses and issues in the datasets. Thus, they are able to improve the datasets to be compliant with the requirement specifications. To illustrate their approach, they develop a ML software for pedestrian detection from the automotive domain. They defined their approach as a process for specifying the requirements in these four major steps:

1. **Benchmark the domain:** The aim of this phase is to analyse the domain problem and define a benchmark ontology to structure the acquired information. This phase is performed in two steps. The first step consists in analysing, identifying and specifying some characteristics that describe the concepts (e.g. recognising pedestrians by a car) to be recognised by a ML software. These characteristics are used to specify the requirements for the corresponding ML software. In order to obtain all these characteristics, the engineer is supposed to perform a web-search and collect all relevant terms of the domain problem. The second step consists in creating an ontology with the gathered information (e.g. characteristics, terms and concepts), which shall be used for the requirement specification. The aim is to visualize the collected information from the web with the ontology. The final ontology shall be reviewed by domain experts to verify of the acquired knowledge about the domain problem is sufficient to build the datasets.

2. **Interpreting the domain in dataset:** The aim of this phase is to understand the collected data in the datasets in order to identify risks, inconsistencies and weaknesses in the dataset. An engineer extracts relevant information and features from the dataset using tools such as Vision API (e.g. tool for analysing images). These features are then analysed and compared with the previously concepts specified in the previous steps. These identified inconsistencied might produce some issues in the dataset leading to inadequate ML models. With the knewly acquired knowledge about the dataset, the engineer build and ontolology to describe the comprehension of the dataset.

3. **Interpreting the domain learned by ML model:** According to authors, interpreting ML models is still an open challenge. The complexity of interpreting ML models differs from model to model. However, the authors present two ways of interpreting such model: using feature extraction techniques to extract relevant properties of a model, or transform ML model into more comprehensible models (e.g. set of logical rules...) with AI techniques. Finally, the engineer build an ontology describing the acquired knowledge of the ML model.

4. **Minding the gap:** During this phase, the engineer compares the acquired knowledge of the domain, the datasets and ML moddel model to identify correlations and gaps among the onologies. Among others, the engineer identifies underspecified concepts, quality concerns, unambiguous requirement specification...

Figure 2.9 shows a high-level overview of the presented approach.

This PhD thesis differs in the sense that we focus on building datasets with precise data selection criteria resulting from a requirements specification. Moreover, we target to engineer a neural network that satisfies the specified requirements. To do so, we iteratively discuss a trained version of a neural network with our customer, update the requirements and augment the dataset with synthetic data. Additionally, we specify the acquired recognition skills of a neural network as key-properties, which are used to verify whether the requirements have been satisfied. We offer a tool to specify requirements and the key-properties using a textual domain-specific language. Our tool supports the SEMKIS methodology and is embedded in our iterative software engineering process during various activities (e.g. pre-activity, analysing the test monitoring data...). Our aim is to iteratively refine the requirement specification to augment the datasets with precisely selected data (compliant with the specification) until the neural network satisfies the requirements and receives the customer's validation.

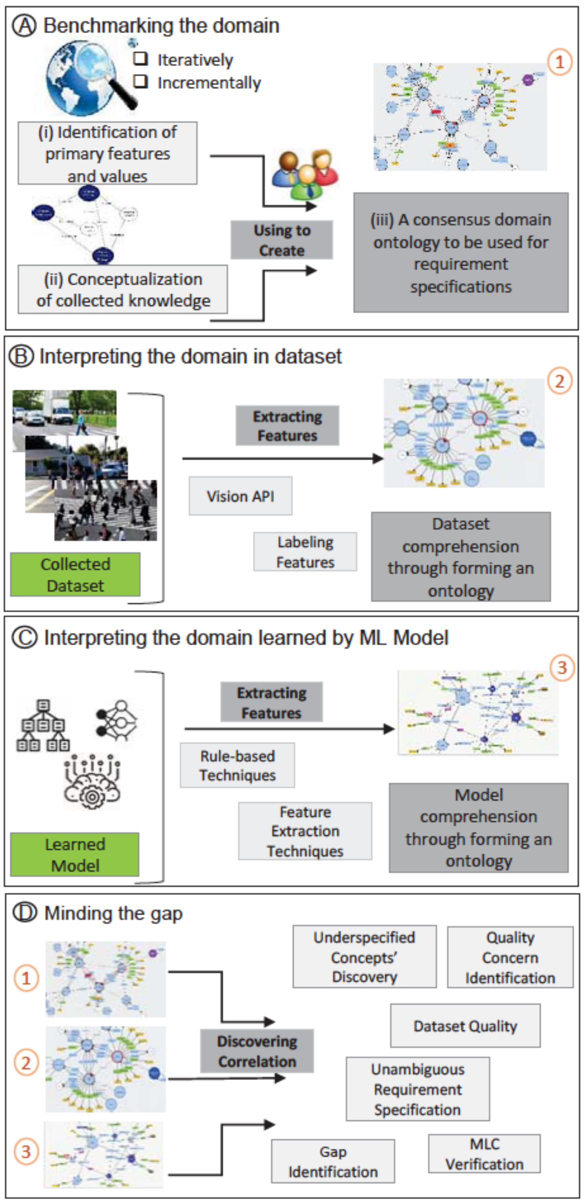

Ⓐ Benchmarking the domain
Iteratively
Incrementally
(i) Identification of primary features and values
(ii) Conceptualization of collected knowledge
Using to Create
(iii) A consensus domain ontology to be used for requirement specifications
1
Ⓑ Interpreting the domain in dataset
Collected Dataset
Extracting Features
Vision API
Labeling Features
Dataset comprehension through forming an ontology
2
Ⓒ Interpreting the domain learned by ML Model
Learned Model
Extracting Features
Rule-based Techniques
Feature Extraction Techniques
Model comprehension through forming an ontology
3
Ⓓ Minding the gap
1
2
3
Discovering Correlation
Underspecified Concepts' Discovery
Quality Concern Identification
Dataset Quality
Unambiguous Requirement Specification
Gap Identification
MLC Verification

Fig. 2.9 Approach high-level overview presented by Rahimi et al.[12].

2.2.2.2 MDE approaches and DSLs supporting neural network engineering

In this section, we present various model-driven engineering approaches and domain-specific languages supporting the engineering of neural networks. In the literature, we found several papers focusing on specifications related to the neural network's architecture, execution/training or mathematical operations.

We found two papers presenting domain-specific languages for specifying neural networks, which focus on the specifications of mathematical operations.

Zhao and Huan [118] present a domain-specific language, called DeepDSL. The aim of the DeepDSL is to optimise the memory consumption and reduce the execution time of computations during an execution of a neural network. DeepDSL supports the specification of a neural network's architecture with high-level mathematical expressions. According to the authors, DeepDSL has been designed to be readable by humans, customizable and maintainable (e.g. debugging easiness). A neural network specified in DeepDSL is a composition of functions, where each function represents a neural network's layer. The functions are performing computations with mathematical tensors to compute the desired output. The DSL is embedded in a general-purpose programming language, called Scala[119]. The DeepDSL's syntax is defined with Scala classes and methods. The authors present various features such as code optimisation, syntax error detection, code generation. The code generator is used to generate Java[41] source code from a DeepDSL specification. Scala is used to compile the neural network, which is specified in DeepDSL into Java source code. The generated Java program is optimised in terms of memory, minimal usage of dependencies and can run on various operating systems such as Windows, OS X and Linus. The Java program can easily be maintained in common IDEs such as Eclipse and IntelliJ. Finally, the authors evaluated the DSL in an experiment using many benchmark convolutional neural networks. The neural networks generated from a DeepDsl specification were able to reduce the runtime costs and memory consumption compared to other deep learning execution libraries such as tensorflow and caffe.

Elango et al. [120] present a domain-specific language, called Diesel. The aim of Diesel is to reduce computational ressources and time for executing a neural network (e.g. training, testing and operation) Diesel is a domain-specific language for specifying mathematical operation (linear algebra, algebraic operation) and neural network operations representing the neural network's architecture. The authors target to specify a neural network precisely with mathematical operations instead of traditional graphs (e.g. directed acyclic graphs). Instead, they target to automate the generation of efficient source code for optimising the runtime of neural networks. Therefore, the Diesel DSL supports also a generator, which takes as input a Diesel specification and generated a CUDA source code (a computing platform for processing software on GPUs). According to the authors, Diesel has been designed to have a user-friendly environment, automatically generates efficient CUDE source code, support parameters adjustements in the generated source code. The authors performed some experiments with the Diesel DSL by comparing them with other CUDA-based standard libraries (e.g. CuBlas, Pascal, Volta). Their results show that the performance (execution speedup) of the generated CUDA source code from the Diesel specification is comparable and in some cases better than the standard libraries.

We found four papers presenting domain-specific languages for specifying machine learning and deep learning software, which focus on the specifications of directed acyclic graphs (e.g. specifying layers, neurons, activation function. . .).

Podobas et al. [121] present a framework, implemented in Python, to support the engineering and deployment neural networks on various high-performance computing platforms. Their framework consists of a domain-specific language, called StreamBrain DSL, for specifying the architecture of a neural network and multiple backends for executing and training neural networks. Their aim is to engineer optimised neural networks executable on various backends in order to reduce the runtime respectively the training time. The StreamBrain DSL is a Keras-like library[44] for specifying a neural network as a sequence of different layers (e.g. convolutional, Denselayer. . .). A layer takes some input, performs some computations and outputs some values. The authors focus hereby on the specification of two-layer neural networks (Bayesian Confidence Propagation Neural Networks BCPNN). However, they claim that the language is extensible for supporting the specification for more sophisticated architectures. The authors present some successful experimentations on the MNIST dataset. They showed a successfull reduction of the runtime and training time of a BCPNN.

Garcia et al. [122] present a model-driven engineering approach for specifying a neural network and generating the corresponding source code for various platforms. The aim of the authors is to facilitate and automatise the engineering of neural networks. Their model-driven engineering approach consists of a textual domain-specific language, called AiDSL, for specifying the architecture of a neural network, and a generator for transforming the specification into some source code. The AiDSL has been developed using Xtext[123]. The DSL supports the specification of architectural componants of a neural network such as input, hidden, output layers, activation function and some architectural parameters. Moreover, the DSL supports the specification of a training process by selecting the training type from an enumeration of standard DL training types as well as the training data. Hereby, the training data is specified as a real-valued matrix and the neural network's targeted output as a real-valued vector. The final specification is then used to generate an executable source code for various platforms. The generator has been developed using the Xtend[123] language, which is part of the Xtext framework. Their approach supports the generation of a neural network implementation in the Encog framework[124] (a Java based machine learning framework). The authors present some rules for mapping a specified AiDSL model into Java source code. Thus, their generator transforms the AiDSL model into an executable Java soruce code. The complete AiIDE, which includes the textual editor for the AiDSL, supports various features such as syntax highlighting, content assistance, validation rules, templates, outline view. . . The authors performed a qualitative and quantitative evaluation of their DSL. Firstly, the authors designed their DSL based on a guideline for developing DSLs of better quality (e.g. readable, maintainable, structured, simple…). Secondly, the authors compared the number of characters, words and lines of code used to implement a neural network architecture in AiDSL and Encog. They present that a specification in the AiDSL requires fewer characters, words and lines of code as implemented in Encog.

There exists some domain-specific languages in the machine learning domain. For example, Tensorflow Eager[125] and OptiML[126] are DSLs for improving the engineering of ML software. The researchers use different approaches[1] But, both target to faciliate the implementation of ML software for their domain experts by not reducing the performance (e.g. recognition skills) of ML software. In this list, we present a summary of some mentioned properties of these DSLs:

- Ease the design and implementation ML architecture.
- Improve the maintainability of ML software (e.g. debugging).
- Automatise the generation of optimised source code.
- Not loosing performance (e.g. recognition skills) of the ML model.
- Flatten the lurning curve of larger and complicated libraries.
- Reduce the execution time of ML software (e.g. training time, testing time or runtime)

In this thesis, we do not study domain-specific languages supporting the specification of the architecture of neural networks. Neither do we consider the specification of mathematical operations nor directed acyclic graphs for defining the architecture. In our work, we study a domain-specific language for specifying precisely the requirements and the acquired recognition skills of a neural network. We target to support software engineers with the DSL to define precise data selection criteria for improving neural networks. Moreover, we target to evaluate a trained neural network by verifying whether the requirements have been satsified. Nevertheless, a possible research field could be to study a generator allowing to generate a neural network architecture from a requirement specification. However, we do not contribute to an automated generation of a neural network architecture.

2.2.2.3 MDE approaches and DSLs supporting dataset engineering

In this section, we present various model-driven engineering approaches and domain-specific languages supporting the engineering of dataset for training and testing neural networks.

[1]**Tensorflow eager** follows the approach to define ML software as imperative programs using Python functions, whereas **OptiML** to support the implementation of ML software.

Since we found a limited number related to dataset engineering to neural network, we extended our search to dataset engineering for machine learning software. We have looked for various studies focusing on the specification of data (e.g. images, videos, numbers, text…), the management of data, the generation of data and datasets.

Our team, Ries, Guelfi and myself [127], has published a model-driven engineering method to improve the requirements engineering phase for building deep learning datasets. The aim is to support deep learning scientists with a precisely defined method to engineer dataset requirements for their DL projects. A neural network acquires its recognition skills through a training process on a training dataset. Therefore, data selection is important to train the neural network to learn exactly the desired recognition skills. Selecting incorrect data might lead to incorrectly trained neural networks with undesired recognition skills. Dataset requirements can help to improve the construction of datasets by reducing costs, detect early errors and prevent errors. Therefore, we introduced a method designed as an iterative dataset requirements elicitation process, to specify the requirements and improve its specification. Our approach is based on the specification of dataset requirements as an executable model, called *dataset requirements concept model (DRCM).* Using MDE techniques, a formal DRCM (FDRCM) skeleton specification is generated from the DRCM, which can be completed by a formal analyst. Thanks to the formalization, the DRCM specification can be interpreted as model executions (set of datatype specification instances). Finally, the model executions are used to validate the DRCM by a customer of the DL project or an analyst. To perform our approach, we defined a process consisting of four different activities:

1. **Model Dataset Structural Requirements:** An analyst specifies a DRCM with a customer for the DL project.

2. **Define Formal Semantics:** A formal language expert generate and completes a FDRCM from the DRCM specification.

3. **Execute the Formal Specification:** A formal language expert uses a formal engine to query the FDRCM and generate a sufficient amount of data specification instances (FDRCM executions) for the next activity.

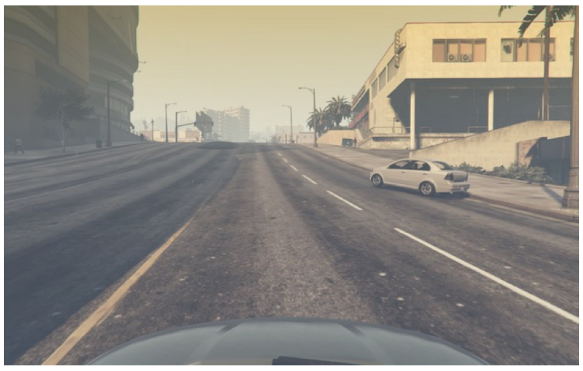
4. **Validation the Dataset Requirements Concept Model:** The generated FDRCM executions are analysed by an analyst to validate the DRCM model. The stakeholders can continue to engineer the desired datasets. If the FDRCM executions are insufficient, then we restart the first activity.

The validated DRCM model represents the validated dataset requirements specification. It shall then be used in further activities to select appropriate data for engineering improved datasets. Thus, the neural network can learn the precisely desired recognition skills with the improved datasets. We also support this process with a tool consisting of four components:

- **DRCM Modeling editor:** A graphical modeling editor (based on the Sirius framework) for specifying a DRCM model with a graphical DSL compliant with UML class diagram syntax.
- **FDRCM Textual editor:** Alloy Analyser for specifying the required formal specification for the FDRCM in the Alloy language.
- **FDRCM Execution:** Kodkod, a SAT-based constraint solver, for generating FDRCM executions.
- **Data specification visualiser:** A custom program for visualising the FDRCM executions.

Finally, we validated our process in an experimentation on the five-segments digit case study. We succesfully used our process to specify and validate some dataset requirements for designing a dataset with five-segments digit images. This thesis differs in the sense that we do not specify only dataset requirements, and we do not specify formal models to use them for validating our requirements specification. We use a textual domain-specific language to specify functional and non-functional requirements of a neural network as well as the neural network's key-properties. The functional requirements serve to specify the targeted data recognition operations, which serve to construct the datasets. The non-functional requirements serve to specify the targeted performance values (e.g. accuracy, loss, recall, precision...), describing metrics for the targeted recognition skills of a neural network. In our method, we specify and validate the requirements directly during multiple meetings and discussion with the customer. We use the requirements to engineer our datasets and our neural network. Afterward the training and testing of the neural network, we specify the neural network's key-properties and compare them with the requirements. During the comparison analysis, we may detect issues in the requirements specification, the datasets or the neural network architecture. In case, the requirements specification requires some updates, we are able to discuss the changes with the customer. Otherwise, we proceed to augmenting the datasets or changing the neural network architecture.

Fremont et al. [13] present a domain-specific language, called Scenic, for specifying objects in a 3D space and generating different scenes (e.g. images, videos). Scenic has been developed to generate synthetic scenes representing the real-world for building cyber-physical systems in the domain of autonomous driving and robotics. For example, it supports the specification and generation of visual scenes of a car driver (e.g. pedestrians crossing the road, cars wrongly parked...). The aim of this domain-specific language is to support engineers to specify the required data, define properties, constraints and policies to generate many scenes for training and testing a neural network. Scenic is a probabilistic programming language, which offers the specification of classes, objects, geometrical information and probability distributions. It offers the specifications of different objects, which can be placed into a 3D space. Objects can be rotated and placed on an absolute position or relative to other objects using a coordinate system. For example, a car A shall be on some position and a car B shall be placed left (or 5 meters left) of car A. Another feature is the specification of a distance distribution (e.g. interval of targeted distances) in between two objects. Thus, it is possible to generate randomly many scenes with different distances in between these two objects. For example, it is possible to generate randomly multiple scenes of two cars on a road, where the distance in between the two car may vary. These properties and constraints can be also combined in the specification for an object. Finally, the Scenic specification is interpreted by analysing the objects, positions and distributions to generate synthetic scenes. To illustrate their DSL the authors present a running example focusing on the generation of car driving scences from a video game, called GTA V. They showed a successfull specification and generate different driving scenes from the video game. Figure 2.10 shows an example of a driving scene and the corresponding Scenic specification.



```
car2 = Car offset by (-10, 10) @ (20, 40), \
           with viewAngle 30 deg
require car2 can see ego
```

Fig. 2.10 Sample driving scene specification using Scenic from Fremont et al.[13].

Other papers [80, 128–131, 86] follow similar approaches. These papers follow mainly approaches focusing on the design of datasets and generating a large amount of different data variants. This thesis differs in the sense that we do not focus on designing concretely datasets and generate data. We want to support the design and construction of datasets with precisely defined requirements. Thus, we define precise data selection criteria to support dataset engineers to build the required datasets.

Finally, there exists various other domain-specific languages in the domain of dataset engineering and deep learning. Many DSLs[132–134] have been developed to support engineers managing any data related to their DL projects. Their syntax is often inspired from popular query languages such as SQL. However, they purpose is often to support engineers to support engineering during different phases of the data management lifecycle. It offers features to use, generate, share and store any kind of data (e.g. datasets, neural network models) for their DL projects. Other researchers study approaches [135] for building datasets for training neural network to automatically generate computer program. These methods are supported by formal domain-specific languages (e.g. DeepCoderDSL[136]) for specifying computer programs as mathematical operations. The formal specifications are then interpreted to generate a dataset consisting of programs, the program's input and output data. The generated datasets are then used to train a neural network to generate automatically a computer program. However, these approaches and domain-specific languages differs from ours in the sense that in SEMKIS, we rely on the specification of requirements and key-properties to build our datasets and neural networks. We use these specifications to determine and define precise data selection criteria to design appropriate datasets and improving neural networks.

Chapter 3

The SEMKIS Dataset Augmentation Process

Abstract

This chapter introduces SEMKIS that we designed as a rigorous process using the BPMN 2.0 modeling language [21]. We present MNIST[137] recognition problem, which we use as running example for illustrating the SEMKIS concepts. The SEMKIS process supports software engineers to build datasets for improving neural networks. Firstly, the data objects required for running the process and its activities, are presented. Secondly, we present the stakeholders, who are responsible to perform the tasks of certain activities in the SEMKIS process. Thirdly, the different process' activities are presented including their input and output data objects and stakeholders. We redefine certain deep learning concepts (e.g. datasets, neural networks, classification, recognition) in the context of software engineering. We introduce new concepts such as the neural network's key-properties and its recognition skills. Key-properties and recognition skills are semi-formal and informal descriptions of the neural network's learned abilities in producing an output given some data inputs, respectively. Finally, we present the data objects, activities, stakeholders on a concrete example of the MNIST recognition problem.

3.1 Introduction

Since many decades ago, researchers and engineers have been working on intelligent machines. Machines equipped with artificial intelligence (AI) based software fsystems have been developed to support humans during their daily activities. These AI-based software systems are supposed to perform advanced tasks, such as perception, communication, speech recognition, decision making,. . . These tasks are considered to be complex and closer to human-level ability. One widely-used AI-technology for building systems that are able to perform such tasks is machine learning (ML). ML-based software systems are algorithms that are able to learn to perform some task from a large amount of data. These systems can acquire the knowledge for performing a complex task through a learning process such as supervised, unsupervised or reinforcement learning [138]. In contrast, traditional software systems consist of manually implemented instructions such as logical rules, loops, computations . . . . Executing a program means to run these instructions in order to produce the program's output. However, the application of ML-based software systems can be very more efficient and simpler to solve certain problems. For instance, it is simpler to develop a machine learning software that is able to recognise various objects in images, than understanding and implementing the required logics for a traditional program doing the same job. In that case, the ML-based software systems learns the logics required to perform the recognition automatically from the dataset. The more the complexity of the desired tasks is increasing and necessites human intelligence, the complexity of the logics required for a traditional program is increasing.

Over the last years, there has been an increasing demand for such machine learning-based software systems. Machine learning is being used in various domains and devices such as cell phones, medicine, finance, education, mobility,. . . These software systems are supposed to automatise complex processes to support humans to perform their activities. One of the most recent ML-methods is deep learning[33] (DL). Deep learning focuses on the study of deep neural networks, which are algorithms representing a simplified model of the human brain. The architecture of a neural network consists of layers, neurons, activation functions, loss function, weights,. . . Like the ML-based algorithms, neural network must learn from a large amount of data to perform a certain task. Therefore, engineers shall build these datasets for training and testing these neural networks.

Data engineers usually build these datasets with some selected data. Given a certain context, a data engineer chooses the required data for training and testing a neural network. For example, images of cats and dogs are selected to train a neural network, which shall learn to recognise them. During the training, the neural network processes the training data and updates its internal parameters of its architecture. After the training, the neural network's recognition is verified using the testing dataset. Depending on the test results, the efficiency of its recognition can be improved by adjusting the neural network's architecture and/or the training data. In this thesis, we focus exclusively on dataset engineering for improving the neural network's recognition.

Data scientist are building these datasets often with randomly collected (or synthetically generated) data by following traditional dataset engineering approaches. These traditional dataset engineering approaches usually consist in collecting, classifying (labelling) and splitting into different types of datasets. Typically, three types of datasets should be built, the training, testing and development datasets. These datasets are required for the development of neural network-based applications. Due to the efficiency of neural networks in many applications, software engineers and data scientists are confronted with the development of more and more neural network-based applications. But, engineers currently rely on traditional dataset engineering approaches to build their handcrafted and empirical datasets for training and testing neural networks.

These traditional approaches are usually very time-consuming and costly. The datasets' sizes are usually very large consisting often of at least 100.000 elements. It is not rare that datasets have millions of elements that must be all collected and classified. The collection and classification of the data is often performed manually. Companies are required to invest a large amount of money for the collection and classification of these data. Additionally, these traditional approaches do not support precise data selection criteria necessary for engineering a neural network that satisfies the customer's requirements. The datasets typically consist of randomly selected data without precise data selection. Since the neural network's recognition depends on the training dataset, the data selection criteria is very important for the training. Due to these traditionally designed datasets, it is very hard to build a neural network that responds exactly to the customers' needs. It is difficult to validate these neural networks, as it is not possible to precisely define the neural network's recognition. In the context of supervised learning, neural networks are trained on classified data[1]. This process may become very slow and costly with an increasing dataset size. It becomes even more complicated, if the dataset must be updated due to neural network's training issue. Since the datasets consist of randomly selected data, there is no specification serving as a base for building or modifying the datasets. Typically, the datasets are shuffled, splited or an amount of randomly selected data is added to the datasets. Engineers hope that the neural network's recognition will improve by following such approaches.

After a deep state-of-the-art research (see section 2.1.2), we have found very few approaches that support software engineers to engineer datasets for improving neural networks. There are even less dataset engineering approaches supporting precise data selection based on customer's requirements. Due to the lack of approaches supporting dataset engineering,

[1]A dataset consisting of labeled information, where the label describes the content of the information

software engineers do not have any guides for building datasets for improving neural network. Most studies focus on modifications of the neural network's architecture to improve its training. We present the main studies in chapter 2. However, our work focuses exclusively on the engineering of datasets to build neural networks satisfying the customer's requirements. Our main motivation is to support software engineers to engineering optimal datasets for improving neural networks in order to satisfy the customer's requirements.

Our research focuses on defining of a new software engineering methodology, called SEMKIS, for engineering deep learning datasets. SEMKIS stands for Software Engineering methodology for the knowledge management of intelligent systems. In our work, we consider as the knowledge, the neural network's skills of recognising anything, and as intelligent systems, the neural networks itself. In this chapter, we focus on the following research question, *what is a software engineering method for building datasets to improve intelligent systems?* We introduce an iterative rigorous process for engineering synthetic datasets in order to improve neural networks and to solve the lack of methods. The process allows to iteratively analyse a trained neural network and update the datasets according to the analysis results. The aim of this process is mainly to support software engineers to develop efficiently datasets for engineering neural networks that satisfy the customer's requirements. Additionally, we would like to reduce complexity of the improvement of neural networks by using synthetically generated datasets.

According to Sommerville[58], software engineering focuses on the development phases of computer program starting from requirements engineering to the maintenance of the final product. The software engineering lifecycle[139] describes the development phases of computer programs. The lifecycle describes phases such as requirements engineering, software design, implementation, testing and the maintenance. Our process shall guide engineers through these phases to develop datasets for engineering improved neural networks by focusing mainly on dataset engineering. Nevertheless, our process has been designed to support not only changes on the datasets but also updates of the neural network's architecture. Our iterative process offers the following support for software engineers:

1. The specification of the requirements used as input for designing and building a dataset and a neural network.
2. A sub-process for analyzing neural network's test results in order to understand the neural network's recognition skills.

3. The notion of key-properties ued for specifying the neural network's recognition skills [2].

4. The specification of a synthetic dataset augmentation used for engineering synthetic datasets to improve neural networks.

5. An iterative improvement of the neural network's architecture or the datasets for training and testing the neural network.

Finally, our process shall support engineers during the discussions of the customer's needs. The final neural network shall be validated and verified with the customer and by taking into consideration his requirements. Our process is useful to support the efficient creation of a dataset based on the requirements coming from a customer. The process eases the interpretation and analysis of a trained neural network with a customer.

In the upcoming sections, we introduce our running examplepresent in detail the SEMKIS process. In the sections3.4 and 3.5, we present the different data objects of our process as well as the stakeholders of our process. Section 3.6 present all activities and subprocesses of the SEMKIS methodology. Additionally, we present an instance of each activity in the context of our running example. Note that the related work can be found in chapter 2.

3.2 MNIST Running example

Before we turn to the actual topic of this chapter, we first present to you our running example. We introduce a running example to provide a better understanding of our work by illustrating our the SEMKIS method in a concrete scenario.

3.2.0.1 Industrial context

We have designed our running example in the context of a software project for a company[3]. Let's pretend that a company has some specific needs for some software to improve an internal working process. In their current process, each employee submits a handwritten work time sheet to the HR secretary at the end of each month. The secretary must read the work time sheets and input the data manually in their salary management tool. However, it happens from time to time that some human-made reading errors occur during the digitalization of the work time sheet. Unfortunately, these errors may potentially lead to some wrongly calculated

[2]**Recognition skills**: the neural network's abilities, which are learned from a dataset during a training phase, to produce the correct/expected output for a given input.

[3]It is important to mention that no real company is involved for this running example. We invented this scenario to design a concrete example that we can use to illustrate our method.

salaries and cost the company a high amount of money. To avoid this error, the company decides to invest in a new software to automate the process of reading the handwritten working hours of their employees. The final software shall improve the company's salary management process and prevent from human-made reading errors.

3.2.0.2 Company's requirements

The company is looking now for software providers, who are engineering an intelligent software system that satisfies their needs. This software shall be designed as a neural network that is capable of recognizing handwritten digits representing the employee's working hours. The aim of this neural network is to transform some handwritten working hours into the corresponding integer value. To reduce the complexity of our running example, we consider that the employees are supposed to declare their working hours only in the range of 0 to 9. Thus, the neural network shall be capable to recognize digits in the range from 0 to 9. It is important to remember at this stage that we use this running example only for illustrative purposes!

A famous problem[140] (MNIST: recognition of handwritten digits) in the domain of computer vision[141] has been considered to design our running example. Since the company requests a neural network for recognizing the employee's working hours, we decided to focus on the problem of the recognition of handwritten digits. However, we only consider the recognition of a single handwritten digit to limit the complexity of the problem. This choice allows us to build a comprehensible running example to illustrate our method.

3.2.0.3 Humans and handwritten digits

Humans write digits in many ways. Their notation usually depend on the human's handwriting and their cultural background. Depending on the region, where humans have lived, they are taught to write the digits in a certain way. For example, the digit '7' is written in several ways like for example '7' with and without a middle dash as illustrated on Figure 3.1. In Luxembourg, a Western European country, 7 with a middle dash, is the preferred notation. In other countries, the digit '7' without a middle dash may be the preferred notation. Even digits, following the same notation, may still differ depending on the human's handwriting even if the people have the same background. However, in the context of this running example, we focus on a neural network shall be trained to recognise exclusively on the used notations in the company.

Due to the high variations of possible digits' notations, humans may encounter some difficulties to recognize the handwritten digits. A software can face similar difficulties

Fig. 3.1 Handwritten '7' with and without middle dash.

to recognize the handwritten digits. Thus, it becomes a challenging task to engineer an appropriate dataset to train a neural network able to recognize correctly handwritten digits.

3.2.0.4 Solution

In this running example, we decided to engineer a neural network to solve the company's problem. The neural network shall satisfy the company's requirements and be able to execute the requested functionality. It shall take as input an image of a digit and output the corresponding integer. In general, neural networks require a large amount of data to learn the desired functionality. Therefore, we selected the MNIST[137] dataset to train our neural network. The MNIST dataset is composed of two sets of labelled images of handwritten digits. The first set is the training set that contains 60000 labelled images. The second set is the testing set that contains 10000 labelled images. Figure 3.2 shows some samples of handwritten digits from the MNIST dataset. The MNIST dataset serves as input for engineering our datasets to build a neural network for the company. In the upcoming sections, we will show many examples illustrating SEMKIS using the MNIST dataset.

3.3 SEMKIS process

In this section, we introduce our iterative process for engineering augmented deep learning datasets. We baptise our methodology **SEMKIS**, standing for **software engineering methodology for knowledge** [4] **management of intelligent systems**[5]. The aim of SEMKIS is to support engineers for the production of intelligent system and analysis its knowledge. Moreover, we target to improve intelligent systems in order to satisfy the requirements of a customer.

The SEMKIS method is designed focusing on the improvement of the recognition skills[6] (an instance of knowledge) of neural networks (an instance of intelligent system). To do

[4]**Knowledge**: the learned abilities of a software system.

[5]**Intelligent system**: a software system that is capable of learning some functionality.

[6]**Recognition skills**: the neural network's abilities, which are learned from a dataset during a training phase, to produce the correct/expected output for a given input.

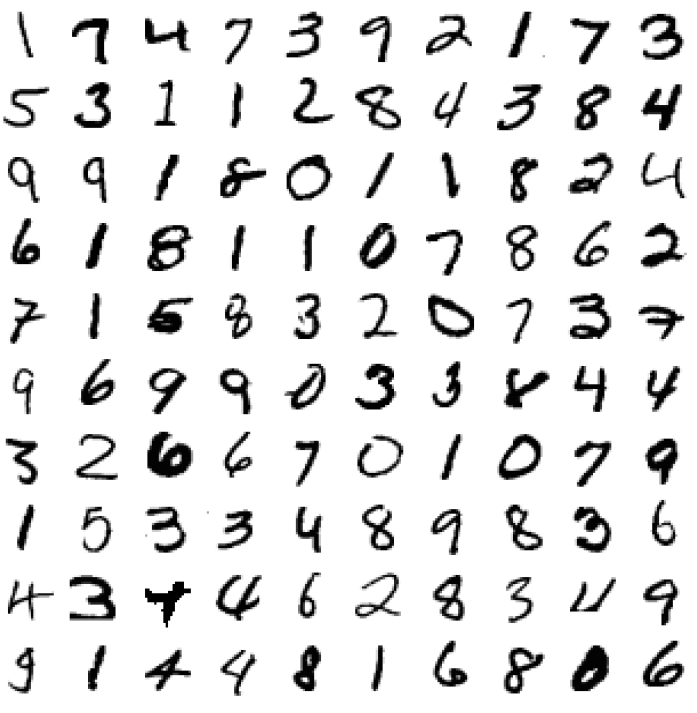


Fig. 3.2 MNIST sample images.

so, SEMKIS mainly focuses on dataset engineering. There exists two ways to improve the recognition skills of neural networks.

- The first possibility is the *modification of its architecture*. Afterwards, the neural network is retrained on the datasets and its recognition skills are verified. This approach can be used if the neural network is not at all capable of learning the expected recognition skills, after multiple dataset modifications.
- The second possibility is the *modification of the datasets* used for training and testing the neural network. This approach can be used if the neural network is not able to improve its recognition skills and the architecture has been already optimised.

In our method, we focus mainly on the modification of the datasets to improve neural networks. To design our method, we applied the concepts defined in the software engineering domain for defining a method for engineering deep learning based software systems. We operate at the intersection of software engineering and deep learning by applying the software engineering concepts to deep learning. Looking at the software engineering lifecycle[58], our method covers different development phases of production of software. In this thesis, we mainly cover the following phases of the software engineering lifecycle:

1. **Requirements engineering**: The aim of this phase is to define precisely the customer's needs in order to build an appropriate dataset for training a neural network that satisfies the customer's requirements.
2. **Design**: The aim of this phase is to define the required data needed for building appropriate datasets in order to train a neural network.
3. **Implementation**: The aim of this phase is to collect and generate data for building the datasets.
4. **Testing**: The aim of this phase is to verify and analyse the recognition skills of a trained neural network for redesigning the datasets and improving neural networks until reaching an acceptable state.

Even though, our primary target was the dataset engineering for improving the recognition skills of neural networks, SEMKIS process includes steps for modifying the neural network's architecture, if necessary. For example in case of a neural network is not able to learn at all the expected recognition skills, the software engineer may modify the neural network's architecture.

SEMKIS has been formalized using an iterative business process language compliant with the BPMN 2.0 [21] modeling language. The aim of this process is to support engineers for building dataset and engineering neural network that satisfy the customer's requirements. The process shall be used to iteratively verify and analyse the recognition skills of a trained neural network in order to generate synthetic data for augmenting datasets. These augmented datasets[7] [57] are used for retraining the neural network and improving its recognition skills. The process provides defined instructions to be followed by software engineers during the development process. SEMKIS shall improve the development of deep learning projects by guiding software engineers for building and modifying datasets, building data as well as building and analysing neural networks.

Figure 3.3 shows an overview of the iterative business process. The SEMKIS process is composed of nine main *activities*, where we defined several activities as sub-processes. The activities are represented as rounded rectangles in different colors (e.g. blue, yellow and green). The process consists of 3 different types of activities (e.g. dataset engineering activities, neural network software engineering activities and neural network execution activities) and 5 different types of data objects (e.g. datasets, neural networks, neural networks data, classified data and augmented data). The engineers require different skills to

[7]Data augmentation in data analysis are techniques used to increase the amount of data by adding slightly modified copies of already existing data or newly created synthetic data from existing data

perform these activities successfully. We describe activity types and the needed skills in the list below :

- **Dataset engineering activities (represented as blue boxes in Fig 3.3):** These activities aim mainly at engineering datasets, building and analysing different kind of data. These activities require skills such as conceptual modelling, statistical modelling and domain expertise. The stakeholder of these activities is usually a data scientist, data engineering or an analyst. Some of these activities are performed with a customer of the neural network in order to define or verify the customer's requirements.

- **Neural network software engineering activities, (represented as yellow boxes in Fig 3.3):** These activities aim at designing and implementing a neural network for a customer. These activities require skills such as programming and knowledge in data mining and mathematics. The stakeholder of these activities is typically a senior software engineer specialized in the development of deep learning-based software systems.

- **Neural network execution activities (represented as green boxes in Fig 3.3):** These activities aim at training a neural network to recognise data, until we obtain the best possible neural network. Additionally, these activities aim at testing a trained neural network, if it is capable of recognising correctly the data. These activities require skills such as knowledge in engineering environments of neural networks (computer hardware, high-performance computing), programming and data management. The stakeholders of these activities are typically software developers or technicians.

There are three different *Data Objects* types used in this process. In Figure 3.3, the Data Objects are represented as rectangles with a folded down corner. They appear in blue, green or yellow, which depends on their nature. Within our process, we have defined these types of Data Objects:

- **Datasets (represented in blue in Fig 3.3).** Datasets are any data needed for engineering neural network-based software systems. They are used mainly for training and testing neural network and for improving other datasets. They serve for acquiring and verifying the capacity of recognizing data such as objects in images, text, voice, objects in videos,...From a mathematical point of view, dataset are usually represented as tensors.

- **Neural Networks (represented in yellow in Fig 3.3).** Neural networks are programs capable of recognising information from data. In our process, we defined a neural network, called *target neural network (targetNN)*. Our goal is to improve the targetNN's recognition skills by augmenting the initial datasets. Moreover, it is possible that multiple neural networks might occur in the process next to the targetNN. For example, a synthesizer neural network might be used to generate synthetic data for augmenting the initial datasets.

- **Neural Networks Data (represented in green).** The neural network's data contain any information resulting from the analysis of the neural network's training or testing. They are usually any kind of monitoring data resulting from these activities. These data contain information such as accuracy, loss, accuracy and loss evolution during the training,...

The Input and Output data objects of the SEMKIS process are classified data respectively augmented data. In Figure 3.3, the Data Objects are represented as rectangles with a folded down corner and a white/black arrow. They appear in white and blue, which depends on their nature.

- **Customer's requirements (represented with an arrow in white in Fig 3.3).** The customer's requirements are defined as an external input of the process and consists of specified needs provided by the customer. Typically, a software engineers discusses the customer's requirements during an initial startup meeting. The provided requirements are specified with a domain-specific language such as our SEMKIS-DSL presented in chapter 4.

- **Classified Data (represented with an arrow in white in Fig 3.3).** Classified Data is defined as an external input of the process and consists typically of a collection of labelled information. A label is typically a one-word description of the associated information. Note that our process focuses mainly on supervised learning[38] problems. However, the process is also compatible with unsupervised or reinforcement learning problems. The terminology might change, but the process shall remain the same.

- **Augmented Data (represented with an arrow in black in Fig 3.3)** Augmented Data is defined as the external output of the process. This Data Object usually consists of augmented datasets, resulting from generated data and the initial classified data.

In the upcoming section, we describe in detail data objects and activities of SEMKIS.

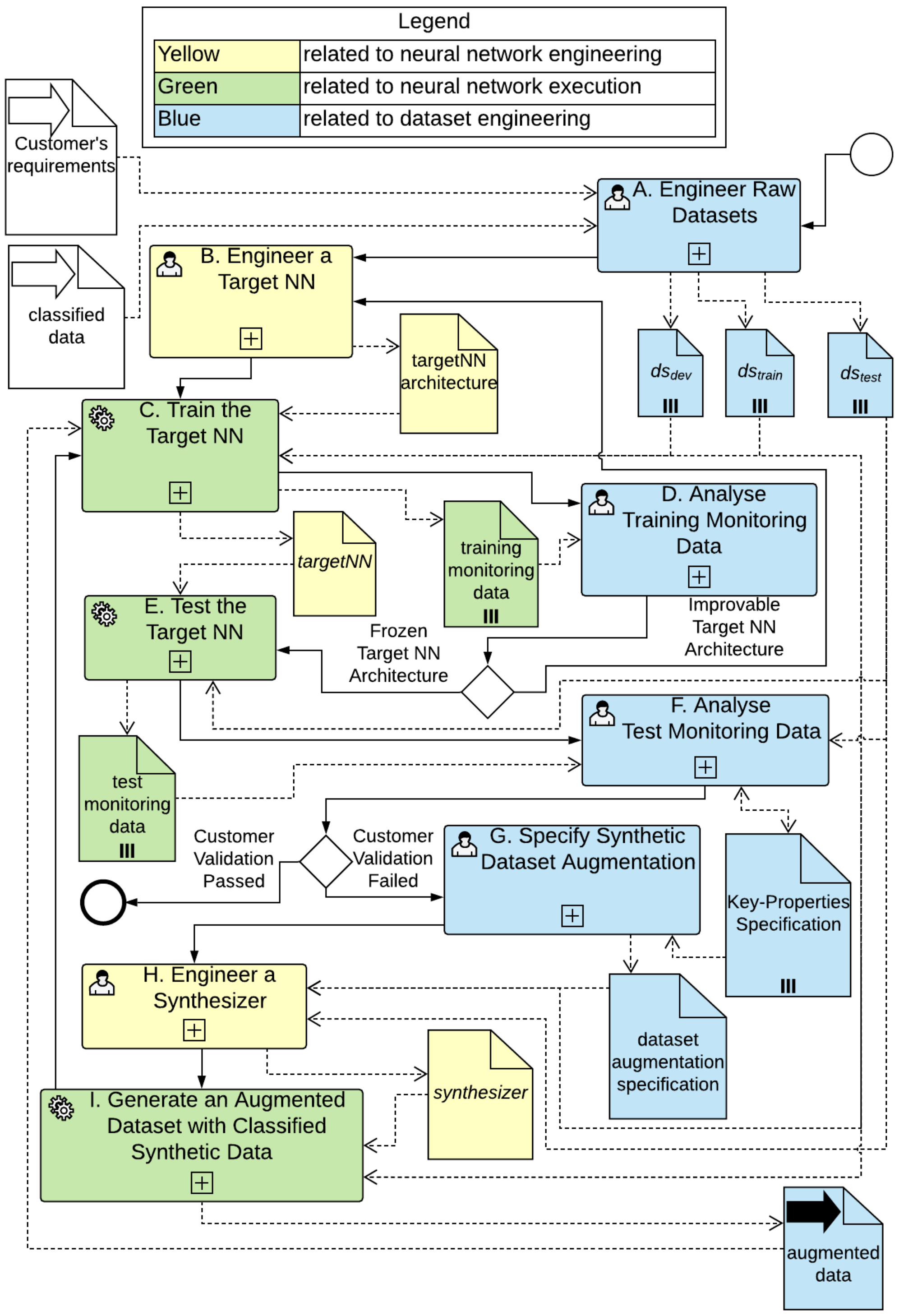

Legend
Yellow
related to neural network engineering
Green
related to neural network execution
Blue
related to dataset engineering
Customer's requirements
classified data
A. Engineer Raw Datasets
B. Engineer a Target NN
targetNN architecture
ds_{dev}
ds_{train}
ds_{test}
C. Train the Target NN
targetNN
training monitoring data
D. Analyse Training Monitoring Data
Improvable Target NN Architecture
E. Test the Target NN
Frozen Target NN Architecture
F. Analyse Test Monitoring Data
test monitoring data
Customer Validation Passed
Customer Validation Failed
G. Specify Synthetic Dataset Augmentation
Key-Properties Specification
H. Engineer a Synthesizer
dataset augmentation specification
synthesizer
I. Generate an Augmented Dataset with Classified Synthetic Data
augmented data

Fig. 3.3 The SEMKIS dataset augmentation process.

3.4 SEMKIS Data objects

In this section, we introduce the data objects of our SEMKIS process. Data objects are any information used as input or produced as output of an activity. These data objects may be datasets, monitoring data, neural networks, documents or any other kind of required information. We present the data objects in the sequential order as they are used in our process.

3.4.1 Data Input

Data is any collection of any information such as images, videos, voice messages, statistics, etc. In our process, we define the **Data Input** as a set of labelled data as we focus mostly on supervised learning problems. Each element consists of an information and a label. The label is a one-word textual description of the associated information. The aim of the Data Input is to provide an initial set of data to build the raw datasets. The raw datasets are different sets of data required during different development phases for engineering a neural network.

3.4.2 Datasets

In our process, we define three types of **raw datasets**, the training dataset, testing dataset and development dataset. The three mentioned datasets contain at least a subset of the Data Input. Among others, these datasets may contain some other data such as collected data or synthetically generated data. However, the datasets shall be distinct (as much as possible) and share a minimal amount of data. This allows us to better test a trained neural network. Especially, the test data shall not be used for training the neural network. All data elements of the datasets shall come from the same domain problem.

- The **training dataset**, ds_{train}, is used for training a neural network. The dataset is used to train the neural network to recognize the dataset's elements. In the context of supervised learning, the datasets consist of labelled dataset elements. During the training, the neural network processes the data elements and learns the associated labels. The dataset must be processed multiple times to learn the associated labels of each data element. The aim of this approach is to train the neural network to output the correct label given a dataset element.

- The **development dataset** [8], ds_{dev}, is used for evaluating a neural network during the training process. During the training, the neural network processes the training data and updates some parameters of its architecture to learn to recognise the data. The development dataset is processed by the neural network after each training round on the training dataset. Since the training and development dataset do not share the same data, the neural network's recognition is evaluated on data that is unknown[9] to the neural network. The development dataset is often a small dataset to minimize the already long waiting times during the training process. The aim of this dataset is to identify first recognition problems of the neural network during the training process. Thus, it is possible to save money (e.g. reduce power consumption, reduce waiting times, shorten training analysis...) if the neural network is not able to learn from the datasets. The goal is to verify whether the neural network is capable of learning to recognise data and to deduce if the neural network's architecture shall be updated.
- The **testing dataset**, ds_{test}, is used for evaluating a trained neural network. During the testing phase, the testing dataset is processed by the neural network after the training of the neural network. It permits identifying additional problems of the trained neural network with more independent data and problems that have not been detected with the development dataset. The testing dataset is typically much larger than the development dataset as it serves to perform a deeper analysis of the acquired recognition skills of the neural network. Therefore, it requires much more samples to precisely test the acquired recognition skills.

3.4.3 Target neural network

3.4.3.1 TargetNN

A **targetNN** is a neural network, which is under development. It is developed by a software engineer and trained on a training dataset and tested on a testing dataset. A targetNN is usually developed for a customer needing a tool for a complex task such as text recognition, speech recognition, image recognition, statistical analysis,... In production, its aim is to recognize similar data to the ones used for training and testing the targetNN.

[8]The **development dataset** is also called the **validation dataset** in the domain of AI/ML/DL. We have chosen the term 'development' dataset, because we use this dataset during the development phase of the neural network to analyse its training. The term 'validation' could be misunderstood as in software engineering validation[58] stands for verifying defines the process of checking a software product satisfying the customer's requirements.

[9]We mean by **unkown**, that the neural network has not been trained on the development dataset. We verify its recognition on this dataset.

3.4.3.2 TargetNN architecture

A **targetNN architecture** is a neural network model such as a convolutional neural network, artificial neural network, generative adversarial neural network, ... The design of a targetNN architecture[142] consists of the number of layers, the layer types, the neurons, the activation functions, loss functions, the weights.... The targetNN architecture is typically implemented by the software engineer.

3.4.4 Monitoring data

3.4.4.1 Training monitoring data

The **training monitoring data** is any information about the neural network's recognition of the training and development dataset gathered during the training. These data may contain any kind of statistics of the recognized and not recognized data. They describe the current state of the targetNN during the training. The aim of the training monitoring data is to provide information about the neural network's training in order to judge whether the neural network is capable of learning to recognise the data or if the architecture shall be updated.

3.4.4.2 Testing monitoring data

The **testing monitoring data** is information about the neural network's recognition gathered after the processing of the testing dataset. These data may contain any kind of statistics of the recognized and not recognized data. The data describe the final state of a trained targetNN after the train and test execution. The aim of the testing monitoring data is to provide information about a trained neural network in order to verify whether the neural network satisfies the expected requirements.

3.4.5 Dataset augmentation specification

In the context of this thesis, a **dataset augmentation specification** is a semi-formal description of several operations for modifying existing datasets. Its aim is to describe precisely the required dataset modifications to improve the neural network recognition. Among others, these modifications may be applied :

- Add real elements to an existing dataset
- Add synthetic elements to an existing dataset
- Modify the elements of an existing dataset

- Remove the elements of an existing dataset

3.4.6 Synthesizer

A **synthesizer** is a program for generating synthetic data. A synthesizer might be designed as a traditional program, a machine learning algorithm (e.g. a neural network). Its aim is to automate the process for generating synthetic data, needed for a dataset augmentation.

3.4.7 Data output - augmented dataset

The **Data output** is typically an augmented dataset. An augmented dataset is a modified version of an input dataset. The modification are usually compliant with the dataset augmentation[57] specification. The datasets serve mainly to improve the training and the testing of a neural network.

3.5 Stakeholders

In this section, we introduce the stakeholders of our SEMKIS process. We define a stakeholder as a person whose activities and interests are of crucial importance for the working environment. In general, a stakeholder of some company might be an employee, a customer, a supplier, a manager ... In this thesis, we have defined three stakeholder categories in the SEMKIS methodology: software engineer, data engineer and customer.

3.5.1 Software engineer

The first stakeholder of our process is the software engineer. The software engineer is responsible managing the different development phase for engineering a neural network based software system. Among others, the software engineer is responsible for these tasks:

- Engineering the requirements to define and specify the customer's needs.
- Designing a neural network architecture.
- Implementing and training a neural network.
- Testing the neural network to verify the satisfaction of the requirements.
- Implementing a program for generating data.
- Executing the program to generate data.

3.5.2 Data engineer or Data scientist

The second stakeholder of our process is the data engineer or the data scientist. The data engineer is responsible for analysing and building the raw dataset as well as analysing and interpreting the training and testing results of a neural network. The data engineer is in charge of:

- Engineering datasets for training and testing neural networks.
- Analysing the test results of neural networks.
- Specifying the recognition skills of neural networks.
- Discussing the development progress of the neural network with the software engineer and the customer.
- Specifying data needed for improving neural networks.
- Engineering dataset for improving trained neural networks.

3.5.3 Customer

The third stakeholder of our process is the customer. The customer is usually the person who needs and orders a neural network. He's responsible for expressing his needs and validating a version of a neural network. The customer is in charge of :

- Expressing his needs as precise as possible.
- Providing eventually some input data for building a training and testing dataset.
- Criticising a version of neural network.
- Validating a trained and tested neural network.

These tasks are performed in multiple meetings with the data engineer and/or with the software engineer. Finally, only the customer is validating a trained neural network to lunch the production of the software.

3.6 Activities

In this section, we present the activities of the SEMKIS process. Figure 3.3 shows a general overview of the SEMKIS process. We decompose the SEMKIS process into its activities and present them in detail in the upcoming subsections. The activities are presented in the sequential order as illustrated in Figure 3.3. We present for each activity the mandatory tasks, that should be completed by a SEMKIS stakeholder. Additionally, we present a concrete instance of each activity in the context of our running example that has been presented in Section 3.2.

3.6.1 Preliminary-Activity - Project settings

Activity description

Before we start with the first activity of the SEMKIS process, we present some preliminary tasks that should be done to set up the project settings.

From a software engineer perspective, this preliminary activity focuses mainly on requirements engineering. During this phase, a software engineer usually meets a customer with specific needs for a neural network. In general, the following steps should be performed during the requirements engineering phase:

1. The customer presents his needs and describes the required software to the software engineer.
2. The software engineer processes the discussed needs and specifies[10] the customer's requirements.
3. The software engineer discusses the specified requirements with the customer.
4. The software engineer updates the requirements based on the discussion with the customer.
5. Repeat steps 3 and 4 until the requirements are stabilized.

Afterwards, the software engineer prepares the required Data Input of the process. The Data Input usually consists of a collection of classified data. Classified data is a set of labelled information[11]. The software engineer collects some data from his customer. These data

[10]The requirements' specification can be formal or informal. We cover the specification task mainly in the next chapter.

[11]A label is typically a one-word description associated with to some information.

shall be used for building datasets to engineer the neural network. Another important aspect, that we would like to emphasize on, is that the customer may provide a dataset for testing the neural network. These data are helpful to construct a testing dataset for verifying and validating the neural network.

Running example - project setup

In Section 3.2, we presented that our case study has a specific need for a neural network. The neural network should be able to recognize handwritten digits that are written by the company's employees.

In the context of our running example, we started our pre-activity by defining and specifying the requirements of the requested intelligent system. The company's project responsible contacts a software company to meet some software engineer. During this meeting, the project responsible discusses the company's needs with the software engineer. The project responsible explains these *needs* :

1. We need a neural network for recognizing the working hours of our employees.
2. Usually, our employees indicate their working hours in the range from 0 to 9.
3. We want to digitalize the working hours into our computers.
4. The neural network should be part of a larger software.

The software engineer analyses the needs to understand the company's problem and specifies the *requirements* illustrated in Table 3.1.

Table 3.1 Functional and non-functional requirements' specification

Functional Requirements	
id	**description**
F-Req1	The neural network shall take as input an image of a handwritten digit.
F-Req2	The neural network shall output the integer value that corresponds to the handwritten digit.
F-Req3	The handwritten digits on the image can be defined in the range from 0 to 9.
F-Req4	The image size shall be fixed to 28×28 pixels.

Non-functional Requirements	
id	**description**
NF-Req1	Neural network shall recognize more than 99.9% of the training data.
NF-Req2	Neural network shall recognize more than 99.4% of the development data.
NF-Req3	Neural network shall recognize more than 99.0% of the testing data.
NF-Req4	Neural network's loss shall be less than 0.05 for all datasets.

Afterwards, the software engineer meets the project responsible. The meeting's goal is to discuss the specified requirements and to obtain the company's collected data. The project responsible validates the requirements and provides two sets of data[12] to the software engineer. The software engineer uses the provided data as the Data Input of our process. In this case, the process' Data Input contains :

1. 60.000 classified images of handwritten digits for neural network training purposes.
2. 20.000 classified images of handwritten digits for neural network testing purposes

Additionally, both sets of classified images do not share common images. Finally, the software engineer proceeds to the first activity of the SEMKIS process.

3.6.2 Activity A - Engineering Raw Datasets

Activity Description

The first activity of our process focuses on *engineering the raw datasets* (A in Figure 3.3). The aim of this activity is to build the required *raw datasets* from the Data Input and to define the *equivalence classes*. We introduce the notion of equivalence classes for the SEMKIS process.

Definition 1 *An equivalence class is a category of related artefacts where each artefact shall be recognised by a neural network within their category.*

To build the raw datasets and to define the equivalence classes, this activity takes as input the *Data Input* and outputs 3 different types of datasets. The *Data Input* consists of a

[12]We suppose that the provided set of data corresponds to the MNIST dataset.

collection of classified data[13]. The main task of this activity is the analysis and the processing of the Data Input by a *data scientist*.

This activity is a subprocess composed of the following four sequential tasks :

1. Create a set of equivalence classes ec_{data}
2. Engineer the training dataset ds_{train}
3. Engineer the development dataset ds_{dev}
4. Engineer the testing datasets ds_{test}

Firstly, a data scientist analyses the Data Input to *create a set of equivalence classes*. The data scientist's main task is to identify categories of related data elements within the Data Input. Each category is a unique class of related data, called equivalence class. Moreover, the data scientist names each category individually. We suggest choosing a unique name per equivalence class. Finally, the name of the equivalence class is typically used to label identically each data element belonging to the same category. In SEMKIS, the set of equivalence classes is denoted as ec_{data}.

Note that the equivalence classes may vary depending on the customer's needs. In that case, the mapping might differ in another problem setting.

The next tasks will focus mainly on engineering the 3 raw datasets. These datasets are called *training, testing and development datasets*. They are composed of a selection of *Input Data*. Usually, these datasets do not share any common data.

Property 1 *The raw datasets, ds_{train}, ds_{test} and ds_{dev}, shall ideally satisfy these properties:*

- $ds_{train} \cap ds_{dev} = \emptyset$
- $ds_{test} \cap ds_{dev} = \emptyset$
- $ds_{train} \cap ds_{test} = \emptyset$

Firstly, the data scientist engineers *the training dataset*. The *training dataset* is composed of a subset of input data. Its optimal size depends on the number of available input data, the computational power of the physical machine and the training process itself. The main purpose of this dataset is to feed the neural network with data to train it to recognize the classes. The training dataset is denoted as ds_{train}.

[13]Classified data is a piece of labeled information. The label is typically a one-word description of the information.

Secondly, the data scientist engineers *the development dataset*. The *development dataset* is composed of an optimal number of data from the input data. The development dataset is usually much smaller than the training dataset to reduce long waiting times during the training. Moreover, it is not much data needed to detect recognition issue tendencies than for a deep recognition skill analysis. Its optimal size depends on the number of available input data and the computational power of the physical machine. The main purpose of this dataset is to observe the evolution of the neural network's learning accuracy and the loss during the training process[14]. The development dataset is denoted as ds_{dev}.

Thirdly, the data scientist *engineers the testing dataset*. The *testing dataset* is composed of a subset of the Input Data. Its optimal size depends only on the available Input Data and the training dataset itself. The main purpose of this dataset is to validate the neural network after the training process. Thus, this dataset should ideally be defined together with the customer as it should serve to verify whether the neural network satisfies the requirements. The testing dataset is denoted as ds_{test}.

Running example - Raw dataset engineering

In the context of our running example, our Data Input is the MNIST dataset, which consists of two sets of classified images of handwritten digits. The first dataset is a set of 60.000 classified images using for building the training and development dataset. The second dataset is a set of 20.000 classified images using for building the testing dataset.

Each image of the MNIST dataset is a matrix of size 28×28. Each monochrome image is associated to a label, being an integer value in [0..9] reflecting the handwritten depiction of the digit, visible on the image. The elements of this image matrix corresponds to a pixel. A pixel is an integer in the range from 0 to 255 representing the grayscale intensity. Thus, MNIST can be defined as $ds_{MNIST} \in \mathscr{P}([0,255]^{28\times 28})$

The next steps focus on the definition of the equivalence classes and the 3 raw datasets. Therefore, we performed our four tasks of this activity:

1. The set of *equivalence classes*, named ec_{MNIST} hereafter, defined as $ec_{MNIST} = \{'0', \ldots, '9'\}$. In the MNIST dataset, there are 10 different labels, which we mapped to our equivalence classes. Each equivalence class represents a label in the MNIST dataset.

[14]The training process is described in subsection 3.6.4 (activity C in figure 3.3)

2. The *training dataset*, $ds_{train} \in \mathscr{P}([0,255]^{28\times28} \times ec_{MNIST})$, contains 54000 images, which are randomly selected from the Data Input's 60000 images. The data selection ratio[15] is 90%.

3. The *development dataset*, $ds_{dev} \in \mathscr{P}([0,255]^{28\times28} \times ec_{MNIST})$, contains 6000 images, which are randomly selected from the Data Input's 60000 images. The data selection ratio[15] is 10%.

4. The *testing dataset*, $ds_{test} \in \mathscr{P}([0,255]^{28\times28} \times ec_{MNIST})$, contains 10000 images, which represent the default testing images of the MNIST dataset.

3.6.3 Activity B - Engineering the targetNN

Activity description

The second activity of our process focuses on engineering a target neural network (targetNN) by a software engineer. The aim of this activity is to choose and engineer an architecture for a neural network that is capable of learning from the training dataset.

The software engineer analyses the equivalence classes and the raw datasets to understand the neural network learning problem (e.g image recognition, speech recognition, language translation,...) and the learning paradigm (supervised, unsupervised, reinforcement learning,....). Other properties that the software engineer includes into his analysis are the format of the datasets, the size of the different datasets, the number of equivalence classes, ... For example, a common neural network architecture for recognizing images is a convolutional neural network [143].

All these properties permit to design and implement a targetNN. Many tasks are necessary to implement trainable targetNN. It includes, among others :

- Selection of the architecture types
- Selection of the type of layers (convolutional, input, output, dropout layer, pooling ...)
- Definition of the number and order of the hidden layers
- Definition of the number of neurons per layer
- Selection of the activation function per layer

[15]Our data selection ration has been chosen such that the development dataset contains a sufficient amount of images for each equivalence class ($\pm$600 images). We have analyzed the images of each equivalence class in the raw datasets to select images with sufficient variations, that we consider as representative for each equivalence class. None of the selected images is part of the training dataset.

- Selection of a loss function, which is necessary for training the targetNN
- Definition of the initial weight for every neuron
- ...

Additionally, it might be necessary to implement a dataset encoder and decoder. This step might be necessary in some cases because the neural network's input is a tensor of real numbers, and it outputs a tensor of real numbers. The aim of the encoder is to build an appropriate readable format of the neural network's input for processing the data through the targetNN. The aim of the decoder is to associate the neural network's output to the targeted equivalence classes.

Running Example - Engineering a convolutional neural network

In the context of our running example, we analyzed the raw datasets and the equivalence classes to determine the learning problem and the learning paradigm[16].

Firstly, we focus on determining the learning paradigm. Our raw datasets consist of classified images, and we target to train our targetNN to learn to recognize the classified data. The targetNN shall learn to recognize the data and map the data to the correct equivalence class. In this case, our learning paradigm is supervised learning, because we train the targetNN to recognize the classified data.

Secondly, we focus on determining the learning problem. Our raw datasets consist of classified images of handwritten digits. We target to recognize these handwritten digits by recognizing the integer value of the digit. Thus, our learning problem is image recognition.

Since we apply supervised learning to train our neural network, and we focus on an image recognition problem, we designed our targetNN as a convolutional neural network (CNN) inspired from Kizhevsyky et al. [144]. We motivate our architecture choice, due to the high ranking of convolutional neural networks for the MNIST recognition proposed by LeCun Y. [137]. Our targetNN is implemented in Python[145] using the Keras Library[44]. The design of our targetNN architecture[17] is illustrated in table 3.2.

[16]We verify if we need a supervised, unsupervised or reinforcement learning approach.

[17]The technical background required for this section can be found in section 2.1.1.

Table 3.2 Our designed convolutional neural network architecture

CNN architecture

id	**layer**	**parameters**	**activation function**	**input shape**	**output shape**	**purpose**
lyr1	*convolutional*	64 filters; (5×5) kernel	*relu*	$(28{\times}28{\times}1)$	$(28{\times}28{\times}64)$	Extracting high-level features[18] from the input data.
lyr2	*convolutional*	64 filters; (5×5) kernel	*relu*	$(28{\times}28{\times}64)$	$(28{\times}28{\times}64)$	Strengthen the extracted high-level features.
lyr3	*max pooling*	(2×2) pool		$(28{\times}28{\times}64)$	$(14{\times}14{\times}64)$	Filter the most relevant features to fasten the training.
lyr4	*dropout*	50% rate		$(14{\times}14{\times}64)$	$(14{\times}14{\times}64)$	Improve training by increasing random selection to avoid overfitting.
lyr5	*convolutional*	64 filters; (3×3) kernel	*relu*	$(14{\times}14{\times}64)$	$(14{\times}14{\times}64)$	Strengthen the extracted and filtered high-level features.
lyr6	*convolutional*	64 filters; (3×3) kernel	*relu*	$(14{\times}14{\times}64)$	$(14{\times}14{\times}64)$	Same as lyr5.
lyr7	*max pooling*	(2×2) pool		$(14{\times}14{\times}64)$	$(7{\times}7{\times}64)$	Same as lyr3.
lyr8	*dropout*	50% rate		$(7{\times}7{\times}64)$	$(7{\times}7{\times}64)$	Same as lyr4.
lyr9	*flatten*	none		$(7{\times}7{\times}64)$	3136	Flattening matrix to array for next layer.
lyr10	*dense*	128 neurons	*relu*	$(7{\times}7{\times}64)$	128	Learns to distinguish the features.
lyr11	*dropout*	50% rate		$(28{\times}28{\times}64)$	128	Same as lyr4.
lyr12	*dense*	10 neurons	*softmax*	$(28{\times}28{\times}64)$	10	Output probability distribution over 10 equivalence classes.

[18]Features are specific characteristic (e.g. shapes, edges, colors, specific image parts. . .) of a data element in a dataset.

We implemented our designed CNN architecture in Python using the Keras library. A code snippet of our code snippet is illustrated in Listing 3.1.

```
def build_cnn(self):
  """
  Builder for a new CNN instance.
  :return:
  """
    self.model.add(Convolution2D(filters=64, kernel_size=(5,5), padding='Same',
        activation='relu', input_shape=(28,28,1)))
    self.model.add(Convolution2D(filters=64, kernel_size=(5, 5), padding='Same',
        activation='relu'))
    self.model.add(MaxPooling2D(pool_size=(2, 2)))
    self.model.add(Dropout(rate=0.5))

    self.model.add(Convolution2D(filters=64, kernel_size=(3, 3), padding='Same',
        activation='relu'))
    self.model.add(Convolution2D(filters=64, kernel_size=(3, 3), padding='Same',
        activation='relu'))
    self.model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
    self.model.add(Dropout(rate=0.5))

    self.model.add(Flatten())
    self.model.add(Dense(128, activation="relu"))
    self.model.add(Dropout(rate=0.5))
    self.model.add(Dense(10, activation="softmax"))

    self.model.compile(optimizer='adam', loss="categorical_crossentropy", metrics=[
        "accuracy"])
```

Listing 3.1 Code Snippet of our CNN implemented in Python using the Keras Library

3.6.4 Activity C - Train the targetNN

Activity description

After having designed and implemented the targetNN's architecture, the software engineer focuses on training the neural network. In this activity, the targetNN processes the two raw datasets, ds_{train} and ds_{dev}. Typically, the software engineer implements some algorithm or uses an existing framework (e.g. tensorflow[46]...) to train the targetNN. Thus, the software engineer shall select the best suitable training method for his type of targetNN.

During the training, the targetNN learns to recognize the training data. The training's goal is to train the targetNN to recognize the whole dataset. To achieve this goal, the targetNN processes an element of ds_{train} and outputs some probability distribution. The distribution contains all probabilities, which describes that a data element belongs to an equivalence class. To interpret this probability distribution, the software engineer must define a recognition rule. The purpose of the recognition rule is to define when an equivalence class is considered as recognized [19] by a targetNN. Therefore, the software engineer defines a probability threshold for filtering the probability distribution and so the equivalence classes. The resulting equivalence classes are considered to be the recognised equivalence classes. A data element is recognised in an equivalence classes if its probability is above 95% in the probability distribution. This probability threshold[20] has been selected after multiple training iterations to:

- Ease the selection of the CNN's recognition certitude, if multiple high probabilities have been outputted by the targetNN.
- Maximise the certitude of the CNN's recognition.

Now, the resulting probability distribution should be compared to the class of the dataset's element. The software engineer shall encode that class (or data label) into a probability distribution, which is comparable to the targetNN's output. In the next step, a loss function should be used to compute the relative error in between the targetNN's output and the encoded the class of the dataset's element. Finally, the loss value can be used to update the targetNN's weights using a gradient descent algorithm [146] and a back-propagation algorithm [147].

This process is repeated multiple times for all data of ds_{train}. Each training iteration on the training dataset is called an epoch. After all training epochs have been completed, the targetNN is considered to be trained on ds_{train}.

The software engineer can use other methods to train the targetNN to recognize the data of ds_{train}. For example, the training dataset can be shuffled after every epoch. This technique avoids that the neural network learns also the 'order' of the dataset and improve the generalization on the dataset. It might be useful to use different gradient descent algorithms for updating the weights, which would improve the targetNN's training.

This might be useful if the targetNN struggles in learning the training dataset. Therefore, the development dataset, ds_{dev}, should be processed after each epoch to observe the neural network's learning curve. The development dataset is used to verify if the neural network overfitting or underfitting [148].

Definition 2 *The training of a targetNN was not successful if the targetNN has at least one of the following properties: ...*

[19]Recognized : a neural network has predicted the expected and correct equivalence class.

[20]Note that the probability threshold depends on the customer's requirements. It can be lower (or higher) for a certain equivalence classes depending on its priority.

- *overfit, meaning if the state of its weights is updated to recognize almost only the data of ds_{train}.*
- *underfit, meaning if it is not possible to update the weight such that the targetNN recognized the ds_{train} and ds_{dev}*

The software engineer can detect signs of overfitting and underfitting during the targetNN's training. These signs can be detected by computing the accuracy and loss on ds_{train} and ds_{dev}. It is quite common that the accuracy and loss are automatically calculated by an algorithm during the training. In both cases, the training might be stopped, if some problems have been detected in order to save waiting time. The reason is that the training time strongly depends on the physical machine and the dataset size. After having stopped the training, the training methods can be improved to retrain the targetNN. If the training has been successful or other methods did not improve the targetNN, the software engineer continues with the next activity.

Running Example - Train the convolutional neural network

So far, we have designed and implemented a CNN architecture. The next activity focuses on training our CNN to recognize handwritten digits.

We use our training dataset of the previous activity to train our CNN. We trained our model on a physical computer with 2 GPUs (Nvidia GTX1080Ti). The execution of the training lasted for ±10 minutes and 50 epochs. Additionally, we split our training dataset into batches of 64 data elements. This allows us to use the mini-batch gradient descent algorithm[149] to update the weights of the CNN during training. We use tensorflow[46] as backend, which includes the gradient descent algorithms, to perform the update of the neural network's weights.

We use the development dataset of the previous activity to observe the training evolution of the CNN. During the training epochs, we observed the accuracy and the loss to detect signs of overfitting. The resulting training/development accuracy and loss evolution is illustrated in Figure 3.4. The accuracy and loss evolution will be analysed in the next section of this chapter.

3.6.5 Activity D - Analyse Train Monitoring Data

Activity description

After having trained the targetNN, the software engineer analyses the results of the training. The training results are called the train monitoring data. The aim of this activity is to verify

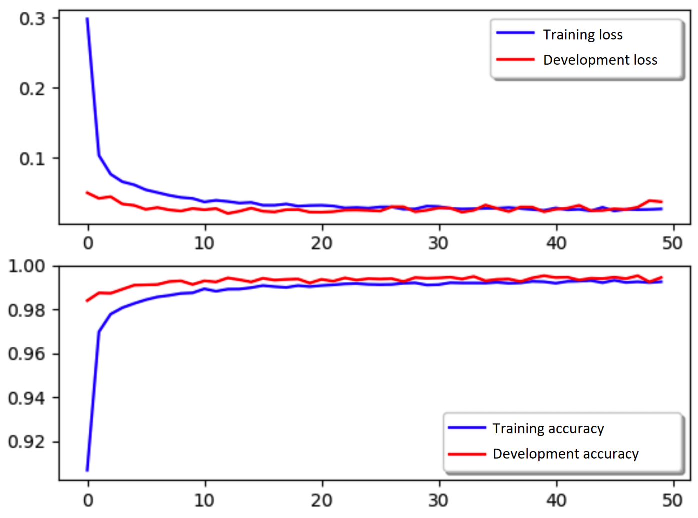


Fig. 3.4 Accuracy and Loss evolution of our CNN.

whether the training has been successful or if it is required to perform some updates on the targetNN's architecture.

Usually, the train monitoring data consists of the following artefacts:

- average accuracy and loss value of ds_{train} and ds_{dev}
- average recognition accuracy value per equivalence class
- evolution of the average and loss in time

The train monitoring data is usually generated automatically by some algorithm. It might be an algorithm from a neural network execution library or some custom algorithm developed by the software engineer.

The *average accuracy and loss value* are calculated during the neural network's training on the training and development dataset. The software engineer shall define a threshold from which the accuracy and loss are considered to be accepted respectively denied. The train monitoring data is used to verify the global targetNN's training. Thus, the software engineer can judge how accurate the targetNN's recognition is and detect potential training problems.

Another train monitoring data is the *average recognition accuracy value per equivalence class*, which is calculated by some algorithm. This accuracy is computed for each equivalence class in each dataset for verifying the targetNN's recognition of the data for the different classes. Similar to the previous train monitoring data, the software engineer shall define a threshold from which the accuracy is considered to be accepted. The property's aim is to detect training problems specifically related to the different equivalence classes.

Finally, the *evolution of the accuracy and loss* shall be computed at the end of each epoch on the training and development dataset. The neural network's accuracy and loss shall be accepted if the threshold has been met. Usually, this evolution is represented as a graph, in which the software engineer compares the accuracies and the losses of both datasets after every epoch. For example, the software engineer can detect if the targetNN does not recognize the elements of ds_{dev}, but it recognizes the elements of ds_{train}. Then, the targetNN tends to overfit. The aim of the evolution of accuracy and loss is to detect signs of overfitting or underfitting.

Nevertheless, if the three properties are satisfied and the neural network's training was successful, the software engineer freezes the architecture and continues with the next activity of our process. Otherwise, the software engineer updates the architecture of our targetNN and relaunches the training again.

It is important to mention that our process allows the modification of the architecture and does not only permit the modifications of datasets to improve neural networks.

Running Example - Analyzing the CNN's train monitoring data

In the context of our running example, we have generated the train monitoring data for analyzing the CNN's training. We have analyzed the following three elements of our train monitoring data to evaluate the targetNN's recognition.

- The *average accuracy and loss on training and development dataset* of the entire training process has been calculated at the end of the training using built-in function of the Keras and Tensorflow library. Table 3.3 illustrates the CNN's average accuracy and loss on the training and development datasets. For both datasets, we observe rather high accuracies and pretty low losses for both datasets. Thus, we conclude that our CNN is not overfitting or underfitting, because it is recognizing pretty well the images of both datasets

- The *evolution of the accuracy and loss graph* has generated using the matplotlib[52] library. We illustrate in figure 3.4 the evolution of the accuracy and loss during the 50 training epochs. We observe that the CNN's recognition accuracy of the training and development dataset improve after each epoch and that both accuracy curves converge close to 100%. The accuracy of the development dataset stays close to the accuracy of the training dataset, which means that the CNN does not tend to overfit. Additionally, we observe that the CNN's recognition loss of the training and development dataset improve after each epoch. Both loss curves converge close to 0. This means that the certitude of the CNN's recognition improves after each epoch on both datasets. Thus, we can confirm that the neural network does not tend to over- or underfit.

- The *average accuracy per equivalence class* has been manually computed by us. We illustrate in table 3.3 the accuracies for each equivalence class for both datasets. We observe that almost all accuracies are over 99% for both datasets. Thus, we accept the data and confirm again that the neural network does not tend to over- or underfit.

In summary, we use these training monitoring data to compare the CNN's recognition (e.g. accuracy and loss) of the development dataset to the training datasets. If the CNN is not able to recognise the development dataset and recognises the training dataset, then it tends to overfit or underfit. However, we can conclude that the CNN is not overfitting or underfitting as it is able to recognise both datasets.

In order to achieve this results, we followed a similar approach to Krizhevsky-et-al [144] by optimising the CNN's architecture in multiple steps. SEMKIS offers the possibility to improve the targetNN's architecture after a unsuccessfull training. As we do not define a formal process for improving the targetNN's architecture[21], the software engineer can chose a preferred approach. We have optimised the CNN's architecture by following a trial-and error approach. Therefore, we iteratively adjusted the following parameters and retrained the CNN in four training rounds, until we reached the specified accuracy and loss:

- the kernel-size on the convolutional layers.
- the dropout probability for the different layers.
- the number of nodes on the first fully connected layer.
- the total number of fully connected hidden layers after the flatten layer.

These parameters have been considered and adjusted to reduce signs of overfitting that have been detected from the accuracy and loss evolution.

Table 3.3 Average training/development accuracy and loss

ds_{train}		**ds_{dev}**	
acc. (%)	**loss**	**acc. (%)**	**loss**

[21]Note that the improvement of the targetNN's architecture is out-of-scope of this thesis as we consider neural network as a blackbox. We focus on the improvement of neural networks with augmented datasets.

99.91	0.0033	99.45	0.0371

Table 3.4 Average training/development accuracy per equivalence class

	ds_{train}			
ec[1]	**imgs[1]**	**cr-imgs[1]**	**icr-imgs[1]**	**acc. (%)[1]**
0	5310	5309	1	99.98%
1	6051	6047	4	99.93%
2	5389	5388	1	99.98%
3	5520	5518	2	99.96%
4	5242	5236	6	99.89%
5	4859	4852	7	99.86%
6	5333	5329	4	99.92%
7	5655	5641	14	99.75%
8	5261	5260	1	99.98%
9	5380	5370	10	99.81%
	ds_{dev}			
0	613	612	1	99.84%
1	691	689	2	99.71%
2	569	566	3	99.47%
3	611	610	1	99.84%
4	600	598	2	99.67%
5	562	557	5	99.11%
6	585	583	2	99.66%
7	610	605	5	99.18%
8	590	588	2	99.66%
9	569	559	10	98.24%

[1]ec: equivalence classes; imgs: number of images; cr-imgs: number of correctly recognised images; icr-imgs: number of incorrectly recognised images; acc.: accuracy in percentage

3.6.6 Activity E - Test the targetNN

Activity description

After having analyzed the train monitoring data, the targetNN shall be tested by the software engineer. The aim of this activity is to perform the required test to verify the targetNN's recognition skills. We introduce the notion of recognition skills that we use in the SEMKIS methodology.

Definition 3 ***Recognition skills*** *are informal textual descriptions of the neural networks learned abilities to map its input data to the correct equivalence class.*

The software engineer uses the testing dataset to verify the targetNN's recognition skills. Therefore, the data is processed through the targetNN, which outputs a probability distribution. As described in Activity D, the probability distribution contains the probabilities that a dataset's element is recognized as being in an equivalence class. The software engineer should use the defined recognition rule to determine the final recognized equivalence class for the processed data.

Once the testing data have been processed by the neural network, the software engineer generates several statistics and figures, containing relevant information about the targetNN's recognition. This generated data is called test monitoring data. Among other, the *test monitoring data* consist of the following elements:

1. *Confusion matrices*[22], used to illustrate the number of correctly and incorrectly recognized images per equivalence class.

2. *Average accuracy and loss.* This accuracy describes the overall percentage of the correctly recognized test data. This loss describes the targetNN's certitude of its overall recognition.

3. *Average accuracy for each equivalence class.* This accuracy describes the percentage of the correctly recognized test data for a single equivalence class.

4. *Grids of correctly and incorrectly recognized images*, used to illustrate and sort the correctly and incorrectly recognized images in a figure.

Finally, the test monitoring data is the output of this activity.

[22]The confusion matrix summarises the number of correctly and incorrectly recognized classes per equivalence classes.

Running Example - Test our convolutional neural network

In the context of our running example, we continued with the testing of our trained CNN. Therefore, we processed the testing dataset to verify the CNN's recognition skills.

Table 3.5 Average test accuracy and loss

$\mathbf{ds_{train}}$	
acc. (%)	**loss**
99.46	0.0251

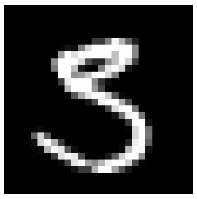
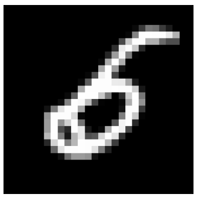
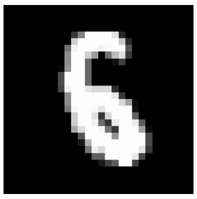
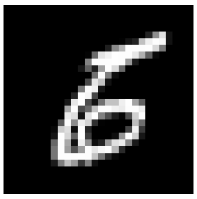
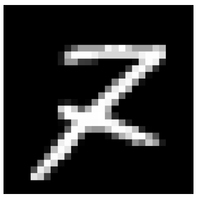
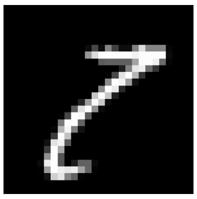
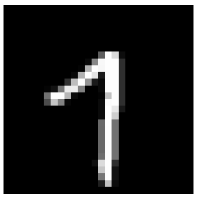
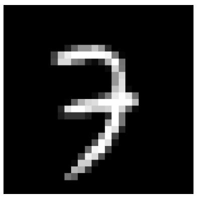
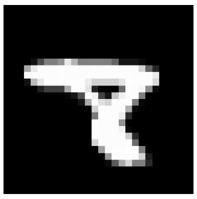
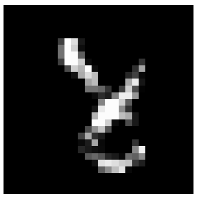
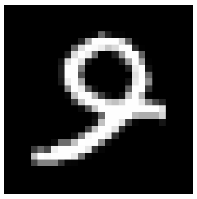
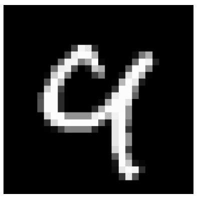
We generated the recommended test monitoring data based on the neural network's output. Table 3.5 shows the average test accuracy and loss of the testing dataset. We achieved an accuracy of 99.46% and a loss of 0.0251. Additionally, we generated a table to illustrate the average accuracy per equivalence class. Table 3.6 shows the resulting average test accuracies for every equivalence class, which have been computed based on the output of the processed test data. Finally, we generated some image grids to represent the correctly and incorrectly recognized images. These image grids will be mainly used and handled in the next activity of our process. Table 3.7 shows a selection of some correctly recognised and incorrectly recognised images

The generated test monitoring data are considered as the output of this activity.

Table 3.6 Average testing accuracy per equivalence class

	$\mathbf{ds_{train}}$			
ec[1]	**imgs**[1]	**cr-imgs**[1]	**icr-imgs**[1]	**acc. (%)**[1]
0	980	978	2	99.80%
1	1135	1134	1	99.91%
2	1032	1027	5	99.52%
3	1010	1007	3	99.70%
4	982	976	6	99.39%
5	892	887	5	99.44%
6	958	950	8	99.16%
7	1028	1019	9	99.12%
8	974	970	4	99.59%
9	1009	999	10	99.01%
Total	10000	9947	53	99.46%

Table 3.7 Samples of incorrectly recognized images

			
$CLASS$[i] $= 4$	$CLASS = 5$	$CLASS = 6$	$CLASS = 6$
$RECO$[ii] $= 9$	$RECO = 6$	$RECO = 5$	$RECO = 8$
P[iii] $= 1.0$	$P = 1.0$	$P = 0.74$	$P = 0.96$
			
$CLASS = 7$	$CLASS = 7$	$CLASS = 7$	$CLASS = 7$
$RECO = 2$	$RECO = 8$	$RECO = 1$	$RECO = 3$
$P = 0.989$	$P = 1.0$	$P = 0.839$	$P = 0.567$
			
$CLASS = 8$	$CLASS = 8$	$CLASS = 9$	$CLASS = 9$
$RECO = 2$	$RECO = 2$	$RECO = 8$	$RECO = 4$
$P = 0.963$	$P = 0.602$	$P = 0.867$	$P = 0.667$

[i] CLASS: classified [ii] RECO: recognized [iii] P: Output probability

3.6.7 Activity F - Analyse Test Monitoring Data

Activity F is a subprocess consisting of four activities for analyzing the targetNN's recognition skills. In this section, we present precisely the subprocess including its activities. In addition to that, we continue to use our running example to illustrate the activities in a concrete scenario.

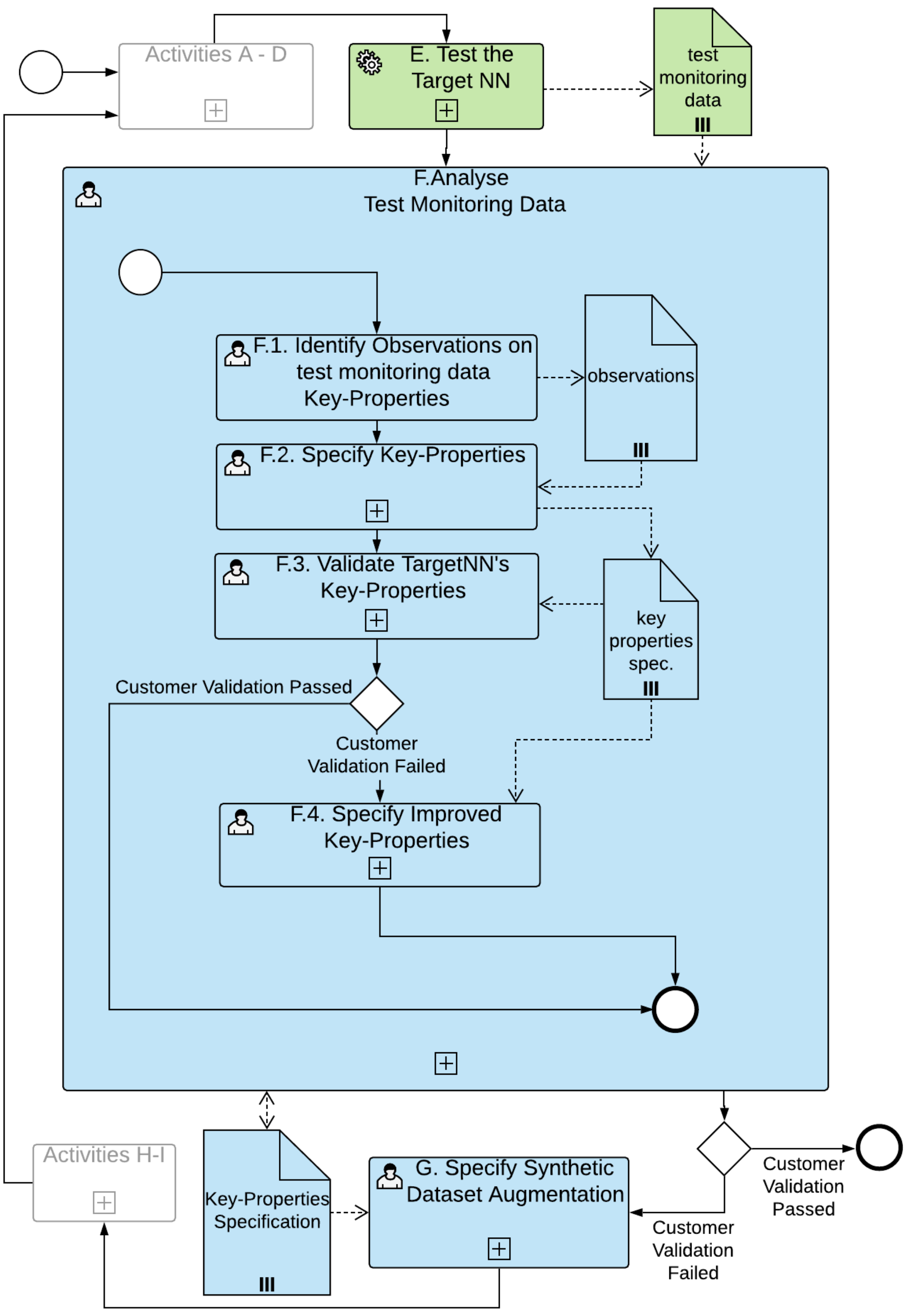

Activities A - D
E. Test the Target NN
test monitoring data
F.Analyse Test Monitoring Data
F.1. Identify Observations on test monitoring data Key-Properties
observations
F.2. Specify Key-Properties
F.3. Validate TargetNN's Key-Properties
key properties spec.
Customer Validation Passed
Customer Validation Failed
F.4. Specify Improved Key-Properties
Activities H-I
Key-Properties Specification
G. Specify Synthetic Dataset Augmentation
Customer Validation Passed
Customer Validation Failed

Fig. 3.5 Test Monitoring Data analysis process.

3.6.7.1 Activity F.1 - Identify observations on test monitoring data

Activity description

In the previous activity, the software engineer had executed the targetNN's tests and generated the test monitoring data serving as input for this activity. The aim of this activity is to specify observations made on the test monitoring data. These observations shall support data engineers to understand and specify the recognition skills of a targetNN.

Definition 4 ***Observations*** *are any information extracted from the test monitoring data, useful for identifying and understanding the recognition skills of a neural network.*

The data engineer's first task is the review of the test monitoring data. We propose to start to categorize the test monitoring data to facilitate the review. The categorization serves for analyzing the test monitoring data with defined objectives instead of blindly reviewing a mass of data. Depending on the amount of data, a blind review may be very time-consuming and complicated. Thus, we suggest defining clear and small objectives to review the test monitoring data. We propose the following categorization of the test monitoring data:

- **Quantitative data** is a subset of the test monitoring data consisting of numerical data.
- **Qualitative data** is a subset of the test monitoring data consisting of textual or boolean data.

The data engineer's second task focuses mainly on analyzing the categorized test monitoring data. During the review, the data engineer extracts relevant information that are required to understand the targetNN's recognition skills These observations describe a group of recognition skills that shall be extracted from the categorized test monitoring data. The data engineer shall specify the observations using a tool of his choice (e.g. tables, text, domain-specific language).

Depending on the test monitoring data, the specified observations shall be grouped into categories and subcategories. Table 3.8 shows some potential categorisation of the observations.

- **Quantitative** observations are represented as a numerical value. (e.g. dataset size, accuracy, loss, number of recognized data,...)
  - **Continuous** observations have their value belonging to a non-countable set. (e.g. accuracy, loss...)
  - **Discrete** observations have their value belonging to a countable set. (e.g. categorisations, number of correctly and incorrectly classified data, dataset size,...)

- **Qualitative** observations are represented as a textual or boolean expression.
  - **Nominal** observations are represented as a textual expression, whose value is a String.(e.g. data description, image content,. . .)
  - **Ordinal** observations are represented as a textual expression in a certain order. Their value is a String being part of an ordered set. (e.g. categorical evaluation of data classification,. . .)
  - **Logical** observations are represented as a logic expression, whose value can be only true or false (e.g. correctness of data classification,. . .)

Table 3.8 Observation categorisation

Obeservation categorisation				
Quantitative (QT)		**Qualitative (QL)**		
Continuous (C)	Discrete (D)	Nominal (N)	Ordinal (O)	Logical (L)

Running Example - Observations for CNN

In the context of our running example, we continue with the analysis of the test monitoring data.

In this activity, we focus on reviewing the test monitoring data to deduce some observations. Therefore, we analysed the test monitoring data resulting from Activity E presented in Section 3.6.6. In addition to the illustrated data, we have analysed the test confusion matrix illustrated in Figure 3.6.

Table 3.9 illustrates the specified observations after our review. We specified the most relevant information needed for determining and understanding the CNN's recognition skills. This specification is used as input in the next activity for specifying the recognition skills.

Table 3.9 List of quantitative and qualitative observations

Quantitative observations		
Category	**Subcategory**	**Observation**
Quantitative	Discrete	Number of CR/IR[23] images
Quantitative	Discrete	Number of CR/IR images per equivalence class

Quantitative	Continuous	CNN's recognition precision
Quantitative	Continuous	Ratio of correctly recognized images

Qualitative observations

Category	Subcategory	Observation
Qualitative	Logical	Image classification correctness
Qualitative	Logical	Image recognition correctness
Qualitative	Nominal	Anomalies in groups of IR images
Qualitative	Nominal	Data consistency
Qualitative	Ordinal	Recognition precision
Qualitative	Ordinal	Recognition policy

3.6.7.2 Activity F.2 - Specify key-properties

Activity description

This activity's input is a specification of observations resulting from the previous activity. In this activity, we analyze the test monitoring data using our previously specified observations. The aim of this activity is to specify precisely the targetNN's recognition from the analysis of the test monitoring data.

The data engineer is expected to analyse the test monitoring data and the categorised observations. After the analysis, the data engineer shall specify the targetNN's recognition skills. We introduce the notion of key-properties to describe formally the targetNN's recognition skills.

Definition 5 ***Key-properties*** *are quantitative and qualitative attributes describing the recognition skills of a neural network. (e.g numerical, textual, boolean,... properties)*

The data engineer uses the observations to define the key-properties that describe the targetNN's recognition skills. Therefore, we suggest that the data engineer performs the following tasks for specifying the key-properties:

1. Select an observation.

2. Select a test monitoring data associated with an observation.

3. Analyse the selected test monitoring data with respect to the specified observation.

[23]CR: correctly recognized; IR: incorrectly recognized

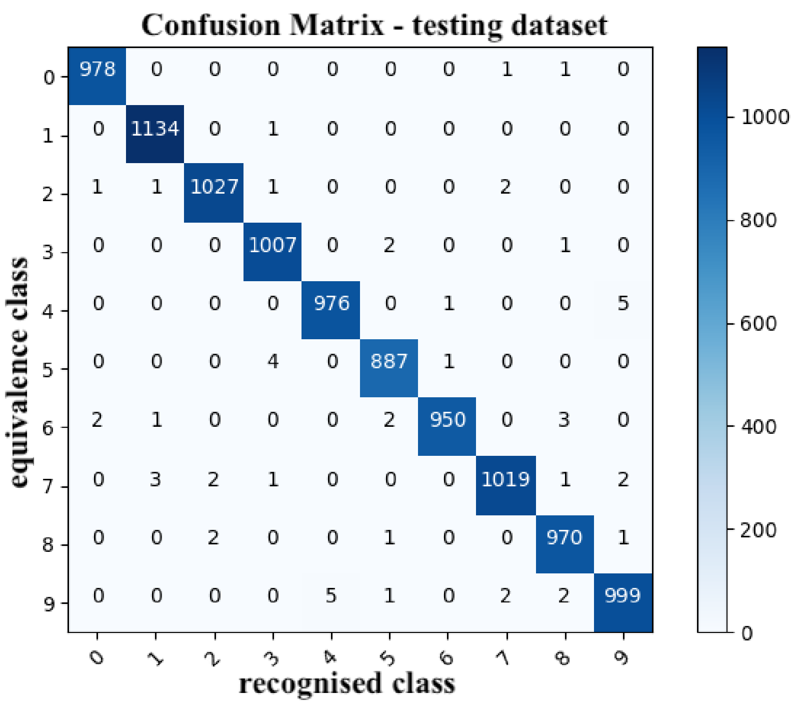


Fig. 3.6 Confusion matrix on the test dataset.

4. Define and specify the key-properties

5. Categorise the key-properties using the categories illustrated in table 3.8

6. Return the first task and continue until all test monitoring data and observations have been analysed.

We propose to categorise the recognized data of the test dataset into four categories: [24].

- Correctly classified and correctly recognized data, denoted as $CLASS \wedge RECO$
- Correctly classified and incorrectly recognized data, denoted as $CLASS \wedge \overline{RECO}$
- Incorrectly classified and correctly recognized data, denoted as $\overline{CLASS} \wedge RECO$
- Incorrectly classified and incorrectly recognized data, denoted as $\overline{CLASS} \wedge \overline{RECO}$

Equivalence classes are categories of related data elements in a dataset and sharing the same label. *Classification* describes the equivalence class in which a data element is categorised.

[24]Note that in machine learning these terms refer to `true/false positive` and `true/false negative`. We have selected the categories `classified` and `recognized` to reduce the amount of mathematical terminology and use more customer-friendly vocabulary. Thus, when discussing the dataset and its labels with the customer, we talk about classifying data. And when talking about the targetNN's input/output relation, we discuss its recogniton with the customer.

Recognition describes the equivalence class in which the data element has been categorised by the targetNN.

These four categories group the testing data based on the correctness of the classification and the recognition. The correctness of the classification is determined by analysing if the different data elements have been categorised in the right equivalence class. The correctness of the recognition is determined by comparing the Data input's classification to the targetNN's recognition.

The data engineer shall specify the key-properties structured similarly to the requirement's specification, such that the key-properties can be compared with the specified requirements to verify their satisfaction. The aim of the key-properties is to allow the verification of the satisfaction of the customer's requirements by the trained targetNN.

The key-properties shall be specified after each process iteration. The specification allows us to keep track of the evolution of the targetNN's key-properties. Thus, the software engineer can improve the visualization of the improvements of the targetNN's in order to satisfy the customer's requirements.

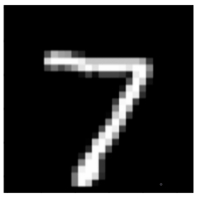
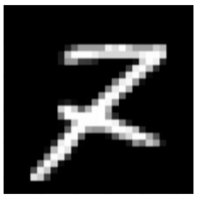
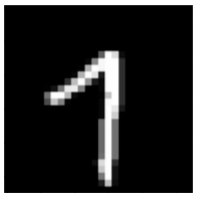
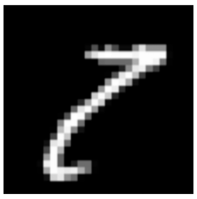
Running Example - Specify CNN's key-properties

We have analysed the test monitoring data and the observations to specify the key-properties of our CNN (our targetNN). First, we sorted the test images into the four proposed categories to ease the analysis. Table 3.10 summarise the results of the correctness of the data's classification and recognition. Table 3.11 shows some samples of categorised images.

Table 3.10 Number of images per category for the classification and recognition correctness

$Class \wedge Reco$	$Class \wedge \overline{Reco}$	$\overline{Class} \wedge Reco$	$\overline{Class} \wedge \overline{Reco}$
9947	35	9	9

Table 3.11 Sample categorisation of images containing a handwritten digit '7'

			
$Class \wedge Reco$	$Class \wedge \overline{Reco}$	$\overline{Class} \wedge Reco$	$\overline{Class} \wedge \overline{Reco}$
$Class_{input} = 7$	$Class_{input} = 7$	$Class_{input} = 7$	$Class_{input} = 7$
$Class_{client} = 7$	$Class_{client} = 7$	$Class_{client} = 1$	$Class_{client} = 2$
$Reco_{NN} = 7$	$Reco_{NN} = 2$	$Reco_{NN} = 1$	$Reco_{NN} = 8$
$P = 1.0$	$P = 0.99$	$P = 0.84$	$P = 1.0$

After the analysis, we specified these key-properties illustrated in table 3.12.

Table 3.12 List of quantitative and qualitative key-properties

Quantitative key-properties		
ID	**Subcategory**	**key-property**
$KP_{v_1,1}$	Discrete	CNN recognised 9947 images correctly and 53 images incorrectly.
$KP_{v_1,2}$	Discrete	CNN did not recognize the digit 7 (9 times) and the digit 9 (10 times) .
$KP_{v_1,3}$	Continuous	CNN's loss on the dataset is 0.0251
$KP_{v_1,4}$	Continuous	CNN's accuracy on the test dataset is 99.47%

Qualitative key-properties		
ID	**Subcategory**	**key-property**
$KP_{v_1,5}$	Logical	CNN recognised the digits in $ec = \{0,1,2,3,4,5,6,8\}$ mostly correctly.
$KP_{v_1,6}$	Logical	CNN recognised the digits in $ec = \{7,9\}$ mostly incorrectly.
$KP_{v_1,7}$	Nominal	The CNN did not recognize mostly a 7 without a middle dash (6 out of 9).
$KP_{v_1,8}$	Nominal	Several images have been wrongly classified.
$KP_{v_1,9}$	Ordinal	The CNN does not recognise the digit 7 very well.
$KP_{v_1,10}$	Ordinal	Select highest CNN's output probability to be the recognized equivalence class.

3.6.7.3 Activity F.3 - Analyse the key-properties

Activity description

In this activity, the data engineer focuses on analysing the specified key-properties of the previous activity. The aim of this activity is to specify the strengths and weaknesses of the targetNN.

The data engineer analyses the specified key-properties of the previous activity. The goal is to describe the recognition skills of the targetNN in natural language. These recognition skills have been described with the notions of strengths and weaknesses. The specification of the strengths and weaknesses shall be presented to the customer of the targetNN. Usually, these specifications shall be comprehensible for a customer, because we make the hypothesis that the customer is not an IT expert.

Definition 6 *A neural network's* ***strength*** *is a textual description of recognition skills in natural language, showing the satisfaction of some customer's requirements.*

Definition 7 *A neural network's* ***weakness*** *is a textual description of recognition skills in natural language, not or partially showing the satisfaction of some customer's requirements.*

The strengths and weaknesses shall be described using a customer-friendly and limited technical vocabulary. These descriptions are less formal than the specification of the key-properties. The specification shall be presentable to the customer of the neural network. We propose that the data engineer discusses the strengths and weaknesses with the customer. The meeting shall be a critical discussion about the targetNN's recognition skills and the satisfaction of the customer's requirements. The goal is to review the requirements' specification and discuss some improvements of the targetNN. The targetNN shall be improved during the next process iteration to satisfy the updated requirements.

If the customer's requirements have been satisfied after the meeting, the targetNN shall be validated by the customer. In that case, the development process is stopped and the targetNN can be deployed.

Running Example - Analyse CNN's key-properties

In the context of our running example, we have analysed the CNN's key-properties to specify a list of strengths and weaknesses of our CNN. Table 3.13 shows a list of weaknesses of our targetNN and proposed solutions. Table 3.14 shows a list of strengths with a description.

We pretend that we have presented the list of strengths and weaknesses to our customer. Let's suppose that the customer proposes that it is important for the software to recognize very well the digit 7 as most of his employees are working in between 6 and 8 hours a day. Thus, the recognition of the digit 7 is critical for the employer.

The issue of recognizing the digit 7 might lead to some severe problems. For the company, it might be a high financial impact, if the digit 7 is not recognized. Thus, we must ensure that the digits in between 6 and 8 are recognized with a high certitude.

Table 3.13 Identified weaknesses at process iteration 1

Weakness	**Improvement proposal**
Critical issues in recognising the digit 7, especially without middle-dash.	Augment the training dataset with images containing the digit 7 with middle dash.
Informative issue of recognising the digit 9.	Augment the training dataset with images containing the digit 9 if customer decided that the improvement is needed.
Priority-based neural network's recognition policy	Define precisely with the customer's priorities on the recognised equivalence classes for setting up a recognition policy.

Table 3.14 Identified strengths at process iteration 1

Strength	**Description**
Recognition correctness	CNN reached an accuracy of 99.47%. CNN recognizes most images correctly
Recognition precision	CNN reached a loss of 0.0251. CNN determines precisely the equivalence class of the input image.

3.6.7.4 Activity F.4 - Specify improved key-properties

Activity description

In this activity, we focus on specifying a list of *improved key-properties*. The aim of this activity is to evaluate the results of the meeting with the customers. The results of the discussion with the customer shall support the data engineer to specify the improved key-properties'. The improved key-properties are used to describe the recognition skills of the targetNN, which should be satisfied during the next iteration. The data engineer uses the improved key-properties to design an improved dataset to train the targetNN. More customer's requirements shall be satisfied by the resulting targetNN.

Definition 8 ***Improved key-properties*** *are formal expressions describing desired recognition skills, which shall be learned by a neural network. (e.g. numerical, textual, boolean,... properties)*

The specification of key-properties shall be used to complete the customer's requirements. Thus, it facilitates to define the scope of a desired targetNN. The aim of this specification is to define clear objectives to improve a trained targetNN. The final output of this activity is the specification of the improved key-properties.

Running Example - Specify improved CNN's key-properties

In the context of our running example, we specified some improved key-properties. These improved key-properties shall be satisfied by our CNN (our targetNN) during the next iteration.

We used the results of the previous activity to specify the improved key-properties illustrated in table 3.15. In the table, we defined the $KP_{v_1,imp,3}$ for introducing an improved key-property. Since the company is located in Europe, most people write the digit 7 with a middle dash. Thus, it is not critical for the company, if an error occur on digits without a middle-dash. The CNN shall be evaluated mainly on handwritten digits with a middle dash.

Additionally, we limit the maximal number of errors that are allowed to occur in the test dataset. The improved key-properties, $KP_{v_1,imp,1}$ and $KP_{v_1,imp,2}$, focus mainly on limiting the number of errors for a specific equivalence class. Finally, we target to update the CNN based on the improved key-properties

Table 3.15 Specification of improved key-properties

ID	KP	description
$KP_{v_1,imp,1}$	$KP_{v_1,7}$	The maximal number of incorrectly recognized images of the digit 7 shall be 4 out of ± 1000 images.
$KP_{v_1,imp,2}$	$KP_{v_1,6}$	Minimize the error occuring for the digit 9.
$KP_{v_1,imp,3}$	$KP_{v_1,6}$	Recognising a 7 without middle dash is optional.

3.6.8 Activity G - Specify Dataset Augmentation

Activity description

In this activity, the data engineer focuses on the specification of a dataset augmentation. The aim of this activity is the definition of a dataset augmentation used for building an improved dataset and retraining the targetNN. The resulting recognition skills of that trained targetNN shall be improved.

The data engineer is in charge of specifying a dataset augmentation. A specification of a dataset augmentation involves the following tasks:

1. Analysing the specification of the key-properties.
2. Analysing the specification of the improved key-properties.
3. Specifying a dataset augmentation for training a targetNN satisfying the improved key-properties.
4. Specifying a similarity function for evaluating the synthetic data.

Definition 9 ***Synthetic data*** *is any information generated with an algorithm used for building dataset to engineer neural networks.*

Definition 10 ***Similarity*** *is a real-valued number in* $[0..1]$ *for expressing the distance in between synthetic data compared to some reference data.*

Definition 11 ***Dataset augmentation*** *is an operation for extending dataset with additional data for improving neural networks.*

There exist various possibilities to modify and augment the training dataset. In addition to the key-properties and improved key-properties specification, we propose that the dataset engineer shall consider the classification and recognition categorisation of the testing dataset before specifying the dataset augmentation. We propose the following list of dataset augmentation operations for each classification and recognition category of the testing dataset. These dataset augmentation operation can be specified to augment the training dataset.

Table 3.16 Dataset augmentation operations for the four classification and recognition categories.

Test data group	Task Description
$Class \wedge Reco$	Generate synthetic data using an algorithm. We propose to strengthen the recognition skills of the targetNN for that category.
$Class \wedge \overline{Reco}$	Evaluate the data to clarify the need of a refinement of the equivalence classes to add some sub-classes. It is required to define the critical equivalence classes for generating synthetic data to strengthen the targetNN's recognition skills.
$\overline{Class} \wedge Reco$ $\overline{Class} \wedge \overline{Reco}$	Parse all these data, move all recognizable data to the correct equivalence class and remove unrecognizable data from the dataset.

Definition 12 ***Unrecognizable data*** *is any wrongly selected and placed information in the raw datasets, which does not satisfy the customer's requirements (e.g. handwritten digits unreadable by humans…).*

Definition 13 ***Recognizable data*** *is any correctly selected and placed informationin the raw datasets, which satisfy the customer's requirements.*

We introduce the notion of data similarity to compare synthetic data to some reference data. Therefore, the data engineer specifies a similarity function to compute the distance in between synthetic and reference data. This function shall have two arguments as input; some reference data and some synthetic data. It outputs a percentage describing the similarity of the input arguments. The function is strongly dependent on the business context.

Finally, the data engineer specifies the dataset augmentation. This specification defines a list of operations to update the datasets. The dataset augmentation specification is used in the next activities to implement an algorithm for generating synthetic data and for generating the synthetic data.

Running Example - specify a concrete dataset augmentation

In the context of our running example, we specified a dataset augmentation to improve the recognition skills of our CNN (our targetNN). We analysed the key-property and improved key-property specification to define a dataset augmentation.

From the key-properties and improves key-properties specification, we determined the following dataset augmentation specification.

Table 3.17 Dataset augmentation specification

ID	description
$ds_{aug,1}$	Generate 5000 random images of a handwritten 7.
$ds_{aug,2}$	Use the structural similarity[150] function to measure the similarity.
$ds_{aug,3}$	Generate the majority of images of a handwritten 7 with a middle-dash.
$ds_{aug,4}$	Generate images of an handwritten 7 with a similarity threshold of $+-80\%$.
$ds_{aug,5}$	Remove the unrecognizable images from the dataset.

Following Table 3.17, we decide to strengthen the recognition of the images in $CLASS \wedge RECO$. Therefore, we target to remove the unrecognizable images to avoid a misleading CNN training. Since the recognition of the handwritten digit 7 is important for the customer, we are required to generate images containing a handwritten digit 7 to improve our CNN. However, most images of a handwritten digit 7 with a middle dash shall be generated. Finally, we use the structural similarity function to verify the synthetically generated data.

It is also possible to define two sub-classes of the equivalence class $\{7\}$. We could define an equivalence class for a 7 with a middle dash (european style) and for a 7 without a middle dash (american-style).

Other examples of dataset augmentation specification could contain the following phrases:

- Move unrecognizable images to a new equivalence classes called 'unrecognizable'. The CNN shall identify the images as unrecognizable.
- Generate a set of 1000, 2000, 3000, 5000, 10000 images of a handwritten 7.
- All generated images must have a minimal similarity of 70%.

Since SEMKIS does not propose a precise method to determine the minimal similarity, it must be determined following a trial and error approach. Therefore, the minimal similarity might be updated multiple times [25] after analysing the generated images. It is required to

[25]Note that this specification shall be compliant with the synthetic data generated by the synthesizer in activity H.

manually verify the generated images in order to assure sufficient variations for training and testing the neural network. For this running example. we have verified the minimal similarities in 10% steps in between 10% and 90%. We have chosen a minimal similarity of 70% because we considered the generated images as appropriate for the datasets after a manual inspection.

3.6.9 Activity H - Engineering a synthesizer

Activity description

In this activity, we focus on engineering a synthesizer for generating synthetic data. The aim of this activity is to develop a synthesizer that meets the specified dataset augmentation.

A software engineer analyses the dataset augmentation specification. Based on the analysis, the engineer decides whether a traditional program or a ML algorithm (e.g. neural network) is needed for the generation of the synthetic data. In both cases, the program shall be engineered to generate the specified data synthetically.

A traditional program may be able to modify the existing data or generate completely new data. The advantages of the traditional synthesizer programs are a full control of the generator, a precise generation of the desired data, and usually faster than neural network-based synthesizers. The engineer can implement a precise and logical algorithm to generate the desired data. There is no training process needed for such program, so the execution time is usually faster for generating a reasonable amount of data. The disadvantages of the traditional synthesizer programs are a slow execution time for generating a large amount of data and an increasing complexity of the program depending on the data to be generated.

A synthesizer neural network (synthesizerNN) supports a more automated process to generate new data. The advantages of a synthesizerNN are a faster generation process of data, a random generation of data variations of the training data and the usability for generating complex data. Once the synthesizerNN has been trained, the program is capable of generating faster a large amount of data. However, the program generates randomly different data variations of the training data. The randomness can be limited to the equivalence classes. Thus, the program can generate random data variations of the training data belonging to an equivalence class. Finally, a synthesizerNN can only be efficient as the training data, on which it has been trained. Thus, the program is capable to generated quite some complex data. The disadvantages of synthesizerNN are a slow training process and the inability of precise data generation. The speed of the training process is highly dependent of the physical machine. Expensive GPUs are needed to accelerate the training speed significantly. Due to the random data generation, this technique is not usable for generating precisely specific data.

Running Example - Engineer a conditional generative adversarial network

In the context of our running example, we have engineered a synthesizerNN for generating our synthetic data. We engineered a conditional Generative Adversarial Network (cGAN) [98] to automate the generation of our synthetic images.

Our cGAN is composed of two neural networks:

- A generator, designed as a convolutional neural network, used for generating synthetic synthetic images of handwritten digits.
- A discriminator, designed as a convolutional neural network, used for determining whether the synthetically generated images are similar to images from the Data input.

The training of these neural networks has some specific goals. The generator shall be trained to generate images, which are similar to the Data input. On the other hand, the discriminator shall be trained to recognize that the generated images are not part of the Data input. After each epoch, the generator improves to generate images similar to the Data input. Meanwhile, the discriminator shall be trained to recognize the generated images as not being part of the Data input. The final training goal is to generate images so similar to the Data input, such that the discriminator must recognize the synthetic images as similar to the Data input.

The cGAN has been trained on 5.000 epochs on our initial training dataset and our 10 equivalence classes. We have computed the similarity of the generated images to the reference images from the Data input using the structural similarity measures (SSM) as specified in the dataset augmentation specification. Its accuracy is computed from the average maximal similarities.

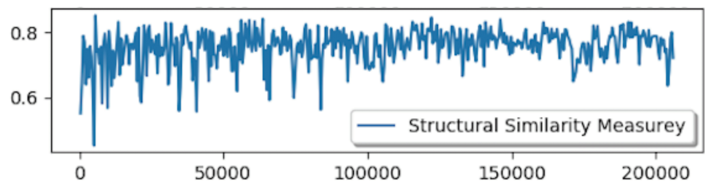


Fig. 3.7 Synthetic image similarity evolution of the cGAN generator

In order to train the generator to generate similar images to the Data input, we trained the cGAN by maximizing our similarity (SSM) during the training. Figure 3.7 shows the evolution of the synthesizerNN's accuracy. The cGAN generator reached an accuracy of 81%.

3.6.10 Activity I - Generate an augmented dataset with classified synthetic data

Activity description

In this activity, we focus on the generation of the synthetic data. The aim of this activity is to execute a synthesizer to generate synthetic data for augmenting the raw datasets.

The synthesizer engineered in Activity H is taken as input in this activity. A software engineer executes the synthesizer. His main tasks consist in monitoring the synthesizer's execution. During the monitoring, he shall focus on these aspectes.

- The synthesizer shall generate synthetic data complying with the dataset augmentation specification.
- The synthetic data shall be evaluated with the specified similarity function.

The generated synthetic data are collected in a new dataset of synthetic data, called ds_{syn}. The dataset contains only synthetic data, which are compliant with the dataset augmentation specification. Moreover, these data shall be evaluated against a certain similarity threshold. The aim is not to introduce new synthetic data that differ too much from the Data Input. We want to generate precisely the data needed for improving the neural network's recognition. The last task of the software engineer consist in building the new augmented training dataset, called ds'_{train}.

Definition 14 *An augmented training dataset is defined as the union of an input training dataset and a synthetic dataset, such that* $ds'_{train} = ds_{train} \cup ds_{syn}$.

Finally, the software engineer can start with the next process iteration. It is important to know that the process also allows modifications of the targetNN's architecture in the next process iterations. If the software engineering encounters major training issues with the new synthetic dataset, he might update the targetNN's architecture. The process allows the iterative modifications of the datasets and the targetNN.

Running Example - Generate synthetic data with cGAN generator

In the context of our running example, we have generated some synthetic images compliant with our dataset augmentation specification. We use our synthesizer neural network, being the generator of our conditional generative adversarial network, to generate the synthetic images.

We executed our synthesizer neural network that generated 5000 synthetic images of the handwritten digit 7. The synthetic images have a similarity of $+-80\%$ to the images of the handwritten digit 7 from the Input Data. Figure 3.8 shows some samples of the generated

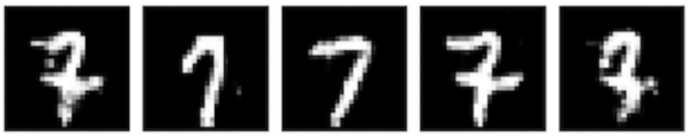


Fig. 3.8 Random samples of generated images of the handwritten digit 7

synthetic images. Thanks the design of our synthesizer neural network the synthetic images are automatically classified. The classified synthetic images are collected together into the synthetic dataset ds_{syn}. Finally, we augmented our training dataset ds_{train} with the data of the synthetic dataset ds_{syn}.

3.6.11 Process iteration - targetNN training with augmented dataset

Activity description

After the synthetic data have been generated, the software engineer restarts the process. The synthetic data are added to the training dataset.

The software engineer performs again the activities from B to F. In summary, the software engineer perfoms the following tasks:

1. The targetNN is trained on the augmented training dataset.
2. The training monitoring data shall be analysed to validate the training and to freeze the architecture.
3. The targetNN is tested on the testing data.
4. The testing monitoring data is analysed to verify if the targetNN satisfies the requirements.

During the analysis of the test monitoring data, the data engineer specifies again the observation to deduce the key-properties. Then, the data engineer specifies the key-properties, the strengths and weaknesses of the targetNN. Again the results are discussed with the customer. If the customer validates the targetNN the process is stopped. Otherwise, the data engineer continues to specify the improved key-properties used for the dataset augmentation. Finally, the dataset augmentation is prepared to start an additional process iteration.

The SEMKIS process is an iterative process. It is repeated until the targetNN has been validated by the customer.

Running Example - Additional process iteration

In the context of our running example, we started an additional process iteration for improving the CNN's recognition skills. Our CNN has been retrained on the augmented dataset, consisting of the old training data and the synthetic data. The training dataset have been augmented with 5000 images of a handwritten digit 7 to improve its recognition.

Table 3.18 shows the average accuracies and losses of the CNN on the training, development and testing dataset. We observe that the overall accuracy is stable for the training and development dataset. The CNN is able to recognize more precisely the development dataset and the testing dataset due to a lower loss value on both datasets. Moreover, the high accuracy shows that the CNN improved its overall recognition of the testing data. It shows that a targetNN trained on augmented data improves its recognition skills.

Table 3.18 Average training/development/testing accuracy and loss

$\mathbf{ds_{train}}$		$\mathbf{ds_{dev}}$		$\mathbf{ds_{test}}$	
acc. (%)	**loss**	**acc. (%)**	**loss**	**acc. (%)**	**loss**
99.90	0.0033	99.42	0.0287	99.60	0.0212

Additionally, we observe that the CNN improved the recognition of the images containing a handwritten digit 7. The accuracy of these images has augmented by 0.5%. The accuracy raised from 99.12% (see Table 3.7) to 99.6%. The targetNN did not recognize 4 images of a handwritten 7, where one image contained a handwritten digit with a middle-dash. Figure 3.9 shows the remaining unrecognized images. Additionally, we observe 10 incorrectly recognized images of the equivalence class '9'. Most errors currently occurred on that equivalence class.

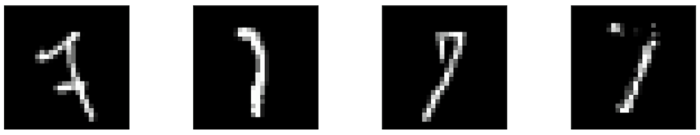


Fig. 3.9 Unrecognized images of the handwritten digit 7 after 2nd process iteration.

Finally, there are many possibilities to continue our process. The customer may validate the current version of the targetNN, because the recognition errors may not have a high impact on the process. Or, the customer may not validate it, which means that a new dataset augmentation shall be performed to iterate the process. Additional changes on the dataset could be:

- Define two sub equivalence classes for a 7 with and without a middle dash.
- Generating random images of the digit 9 to strengthen the CNN's recognition.
- ...

The process shall continue until the customer's requirements are satisfied and the customer validates the targetNN.

3.7 Publications

Our initial work has been peer-reviewed, presented and published at the IEEE International conference in software engineering and service sciences (ICSESS 2019) and the European Symposiums on Software Engineering (ESSE 2020). In these papers, we present the entire process, activities and sub-processes treated in this chapter.

- The paper titled 'Software Engineering for Dataset Augmentation using Generative Adversarial Networks' [3], which introduces the overall process and the different activities.
- The paper titled 'Specifying key-properties to improve the recognition skills of neural networks' [4], which introduces the notion of the neural network's key-properties and the analysis of the neural network's test results.

This chapter presents briefly a synthesized and fine-tuned version of SEMKIS that has been published in both papers.

Chapter 4

SEMKIS-DSL: DSL for the analysis and design of deep learning datasets

Abstract

This chapter introduces a new language, called SEMKIS-DSL, used within the SEMKIS process to specify the requirements and key-properties of a neural network. The main goal of the DSL is to support engineers with structured specification to define an appropriate dataset augmentation to modify datasets for improving neural networks. The SEMKIS-DSL shall support for software engineers for better understanding of the neural network's recognition skills in order to verify whether the requirements are satisfied (or not) and to determine which dataset changes are required for improving the neural network. We present the conceptual model and the concrete syntax of the SEMKIS DSL in this chapter. Then, we present a manual model transformation method for analysing these specifications and specifying a dataset augmentation. We illustrate the concepts of the SEMKIS-DSL on the thesis' running example, called MNIST, focusing on a neural network for recognising handwritten digits. Finally, we present our tool support which can be used to write specifications with the SEMKIS-DSL.

4.1 Introduction

In the previous chapter, we have introduced a process for augmenting datasets to improve neural networks. In order to support software engineers during the preliminary and test monitoriting data analysis activity (see sections 3.6.1 and 3.6.7), we introduce the SEMKIS-DSL, a domain-specific language for the specification of the neural network's requirements and key-properties.

Requirements specifications are needed to understand and define the customer's needs for developing an appropriate neural network. Key-properties specifications are needed to understand the acquired recognition skills of a neural network. In combination, both are needed to analyse and evaluate the learned recognition skills of a neural network in order to verify the satisfaction of the requirements. Depending on the analysis results, a data engineer deduces a dataset augmentation specification to engineer appropriate datasets to improve neural networks. We haven't introduced any tool yet supporting these specifications to facilite the analysis.

Currently, these requirements and key-properties have to be specified manually with classical tools (e.g. Word, Excel...). These classical tools do not offer features like auto-completion, code templates, consistent vocabulary, grammar rules... Moreover, it is complicated to develop algorithms that analyse textual specifications in natural language, as they are often ambiguous. Without precisely defined concepts and a given structure specifications often tend to be incomplete, inconsistent and difficult to read. That is the reason, why requirements and key-properties specifications in natural language are hard to be maintained and error-prone. However, formal specifications can be useful for improving the analysis of the requirements satisfaction, and so improving also the specification of a dataset augmentation to engineer appropriate datasets.

These specifications are analysed by a data engineer, who proposes the dataset augmentation specification for improving the neural network. This analysis is a quite complex task, which is typically performed manually in textual documents. Due to the above described negative aspects of textual specification, it is often difficult to obtain reliable analysis results. Therefore, it is difficult to define a process for analysing these requirements and key-properties in order to specify a dataset augmentation. These issues might lead to incomplete, misleading or incorrect dataset augmentation specifications resulting into multiple process iterations to improve the neural network. We think that engineers require support to improve their requirements and key-properties specification in order to propose a dataset augmentation specification.

The above listed issues motivated us in our research to focus on the specification of requirements and key-properties for neural network based systems and a method for specifying a dataset augmentation to improve neural networks. In this chapter, our research focuses on the following research questions:

- What is a specification of the requirements and key-properties for neural networks?

- What is a method for specifying a dataset augmentation from a requirements and key-properties specification to improve neural networks?

Our research focuses on methods and tools for specifying requirements and key-properties, as well as for analysing these specifications for augmenting and rengineering datasets to improve neural networks.

A possible mechanism for specifying requirements and key-properties is the usage of domain-specific languages. Domain-specific languages [151] are specialized languages used for solving a problem in a specific domain. Software engineers typically design these languages by developing some grammar and defining semantics. These languages are used to describe models with a limited vocabulary at a certain level of abstraction. Typically, the vocabulary chosen for defining the languages' key-words is mostly built of concepts that are used in the domain-problem. Domain-specific language are designed to support domain-experts to perform their tasks without necessarily being computer science experts. The domain-experts usually write in their DSLs using textual editor with various features (e.g. autocompletion, templates, error detection). The development environment, in which the textual editor is used, is usually designed for domain-experts in which they can focus only on their domain-problems. Our domain-problem is the specification of the requirements and key-properties of a neural network.

Domain-specific languages support domain-experts to perform their tasks. In our case, a domain-specific language shall support software engineers to specify the neural network's requirements and key-properties. The resulting specification shall be a structured, readable and complete specification. Moreover, the specification with the domain-specific language shall facilitate the process of defining a dataset augmentation. The usage of a domain-specific language permits the definition of a method for translating the specification of the requirements and key-properties into a dataset augmentation specification. Ideally, the method shall serve as a base for automatising that process and proposing a model transformation.

In this chapter, we present a novel domain-specific language, called SEMKIS-DSL, for the specification of the neural network's requirements and key-properties. Our domain-specific language can be used in various activities of the SEMKIS process to specify the requirements and the neural network's key-properties. Among others, the most interesting activities, where the DSL and the specifications are used, are during 'the preliminary activity', 'dataset engineering activity', 'the test monitoring data analysis', and 'the dataset augmentation specification'. In addition to the SEMKIS-DSL, we propose a new method for analysing this specification and defining a dataset augmentation. The resulting dataset augmentation specification is used to reengineer the initial dataset (training, development and testing dataset). We introduced these modified dataset, called augmented datasets (see section

3.6.10), in the SEMKIS-Process. The augmented datasets are then used to retrain the neural network and successfully improve its recognition skills.

Finally, this chapter presents different aspects required for defining a domain-specific language and a method for defining a dataset augmentation. Section 4.2, we present our conceptual model of the SEMKIS-DSL. Section 4.3, we present our concrete syntax of the SEMKIS-DSL developed using Xtext [123]. We also present the tool shortly in section 4.4.

4.2 Conceptual Model

We have analysed the SEMKIS process to identify the most relevant concepts required for the specification of the neural network's requirements and key-properties[1]. We target to support software engineers with a specification language during their deep learning developments projects. Among all the concepts defined in the SEMKIS process, we have identified these concepts that are necessary for specifying the key-properties of the neural network:

- concepts related to the customer's requirements, required for defining the needs of software customers.
- concepts related to the key-properties, required for describing the neural network's recognition skills.

In this section, we present the conceptual model of our SEMKIS domain-specific language (SEMKIS-DSL), a language for the specification of the neural network's key-properties.

4.2.1 Requirements specification

The first set of SEMKIS-DSL concepts, presented in this section, are related to the specification of the requirements.

Looking at the software engineering lifecycle, the requirement engineering phase is typically defined as the starting phase. During this phase, the software engineer is supposed to discuss the customer's needs. These needs are required to understand and develop the targeted software for the customer.

In traditional software systems engineering, we think that requirements engineering is important for the development of deep-learning based software systems. It is necessary to define and understand these requirements in order to build any kind of software that shall satisfy his needs. Requirements serve to understand the needs of a customer in order to have

[1]The neural network's key-properties describe typically the acquired recognition skills of a neural network after a training process.

a realiable starting point to design and implement the software. Additionally, requirements serve to evaluate and validate the developed software. These requirements become mandatory if we want to verify whether the engineered software satisfy the customer's needs.

In the context of our SEMKIS process, we think that requirements are becoming essential to engineer deep learning-based software systems. Similar to traditional software systems, requirements are important for engineering datasets and neural networks that shall satisfy to the customer's needs. Additionally, they serve for evaluating and validating a version of a trained neural network. In the context, we have identified the following important tasks for engineering requirements:

- Identification of the required data and equivalence classes for training the neural network.
- Selection and Design of a neural network architecture capable of learning from the data.
- Evaluation of the recognition skills of a trained neural network for verifying if the requirement have been satisfied.

The SEMKIS-DSL targets to facilitate these tasks and to support software engineering during the specification of the requirements. In our DSL, the specification of the requirements serve to describe the customer's needs for a neural network. We use the requirements to engineer dataset for improving neural networks. These requirements are used as input in the SEMKIS process for engineering the datasets and the target neural network. In addition to that, we use them to verify if the neural network has learned the required recognition skills Figure 4.1 shows all concepts related to the specification of the requirements.

We defined two types of requirements (**ctRequirements**) that can be specified with our domain-specific language.

- **Functional requirements (FRs) (ctFunctionalRequirement)** describe all required features and functions of a system. These features and functions serve to enable the system and users to perform their desired tasks. In general, functional requirements describe the behaviour of the neural network. It describes the neural network's output given a certain input. In SEMKIS, functional requirements describe the targeted recognition skills of a neural network. They describe what the neural network shall be able to recognize.
- **Non-functional requirements (NFRs) (ctNonFunctionalRequirement)** describe the system's attributes such as performance, reliability, maintainability, scalability,

and usability. In general, non-functional requirements typically describe the quality level parameters of the neural network's recognition such as accuracy, loss, recall, precision. . .

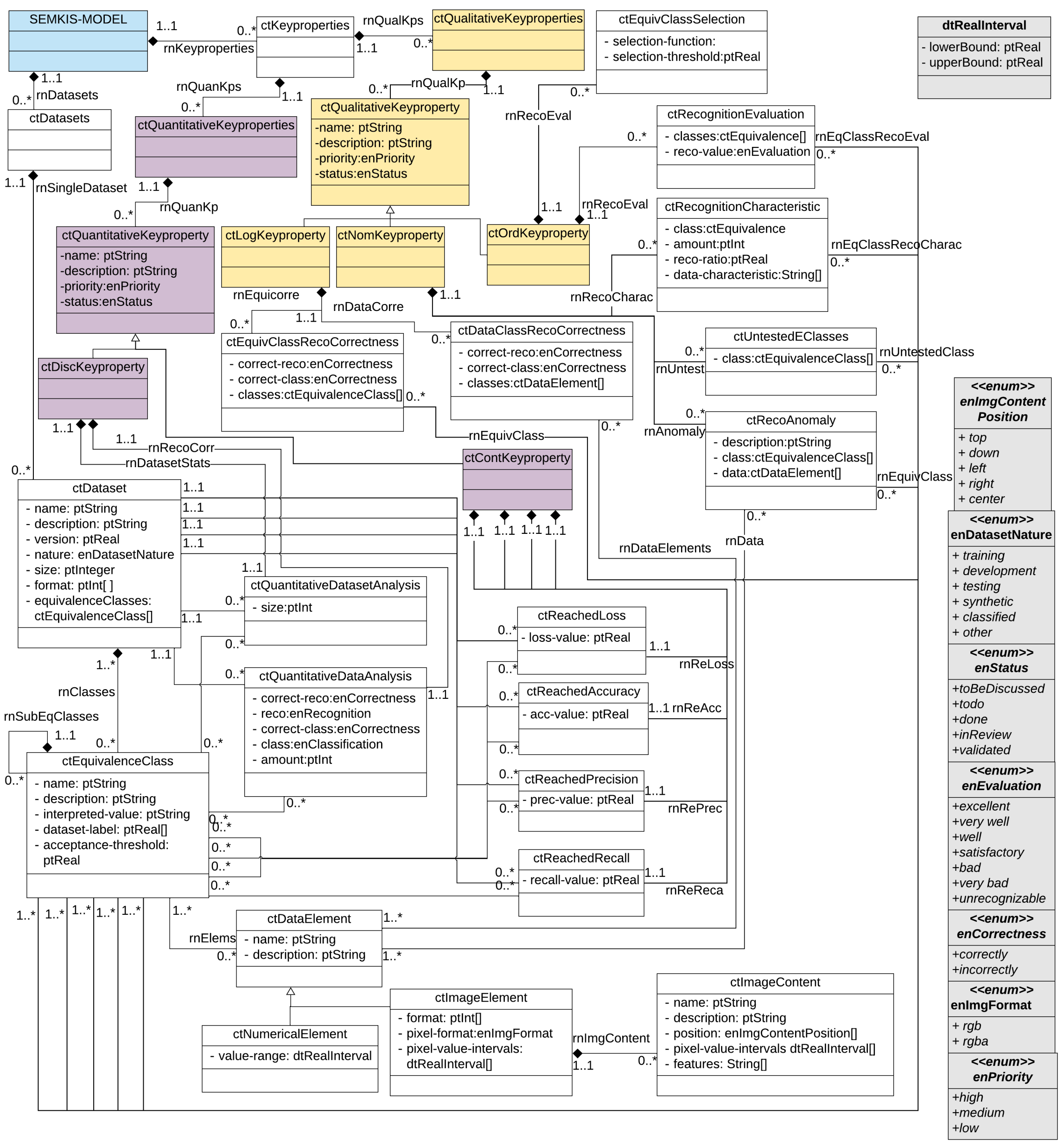


Fig. 4.1 SEMKIS-DSL Metamodel: Requirements specification concepts.

First, we start with the **non-functional requirements (ctNonFunctionalRequirement**). We identified four main types of non-functional requirements that shall be specified with our domain-specific language. We have selected only a subset of often used and known types of non-functional requirements, which are sufficient to introduce the conceptual model of the SEMKIS-DSL and to illustrate the concepts in our case study. However, the SEMKIS-DSL has been designed to easily add other types of non-functional requirements (e.g. AUC-ROC or F1-score) similar to our proposal. These requirements describe mainly the quality level of the neural network's recognition skills on the different datasets.

- **Accuracy (ctTargetAccuracy)** describes the percentage of the all correctly recognized[2] data. For example, the accuracy may be defined by the percentage of recognized data from an entire dataset, many datasets or a single equivalence class.
- **Loss (ctTargetLoss)** describes the recognition error of the neural network. The error is calculated with the neural network's output and the expected output.
- **Precision (ctTargetPrecision)** describes the percentage of the correctly classified[3] and correctly recognized data among all correctly recognized data. Similar to the accuracy, the precision may be computed out of the recognized data from an entire dataset, many datasets of a single equivalence class.
- **Recall (ctTargetRecall)** describes the percentage of the correctly classified and correctly recognized data among all the correctly as well as incorrectly classified and recognized data.

Secondly, we continue with presentation of **the functional requirements**. Since functional requirements describe the behaviour of a software system, we have identified two types of functional requirements. In our context, the behaviour of a neural network can be expressed by describing the expected neural network's output for a given input data element. Hence, the functional requirements describe the required input and the expected output of the target neural network (targetNN). Similar to the non-functional requirements, the SEMKIS-DSL has been designed to be extensible such that other kind of functional requirements can be easily added. We have selected only a subset of types of functional requirements, which are sufficient to introduce the conceptual model and to illustrate the concepts in our case study. The main goal is to describe the targeted recognition of the neural network.

- **TargetNN input (ctTargetNNInput)** describes the required input data of the targetNN. These requirements may describe numerical values, images (and its content), a voice or any kind of information readable by a neural network. In our DSL, we focus

[2]Recognized data are corresponds to the output of the neural network, whether it has produced the correct or incorrect output.

[3]Classified data corresponds to the classifications (labels) of the data in the datasets.

mainly on numerical values and images. However, the DSL can be easily extended by adding the required concepts for describing voice or any other input data element.

- **TargetNN output (ctTargetNNOutput)** describes the expected output of the targetNN. Similar to the input data, they serve to describe any kind of information that has been output by the targetNN. These requirements may describe attributes such as the targeted output, the tolerance range of the recognition, the critical recognitions…

Finally, these specified requirements will afterwards be used to verify, if the targetNN actually performs as expected.

4.2.2 Key-Properties specification

In this section, we present the next set of concepts related to the specification of the key-properties and supported by our SEMKIS-DSL. In traditional software engineering, a software engineer runs typically a test phase to verify the developed software whether it satisfies the specified requirements. During this phase, the software engineer identifies and detects potential system errors, faults or failures. His final goal is to find enough software issues that can be fixed by the developers in order to update and improve the software.

In our method, we follow a similar approach in order to improve a targetNN. It is required to identify the neural network's recognition skills for verifying if it met the specified requirements. We propose a domain-specific language to support software engineers to specify the recognition skills and facilitate the comparison of the identified recognition skills with the requirements. The SEMKIS-DSL permits the evaluation of a trained targetNN and update the datasets for improving afterwards the targetNN.

In our method, we follow an approach that is similar to classical black-box testing (also called specification-based testing) in traditional software engineering. In the context of neural networks, we use a testing dataset to verify whether the requirements have been satisfied. Ideally, the customer shall provide such a testing dataset to verify the neural network. Then, the software engineer's role would be to verify whether the testing dataset is appropriate to test the neural network. If there is not testing dataset, a software engineer shall design an approparaite dataset to test the neural network.

A targetNN shall be tested on a testing dataset in order to obtain a set of test monitoring data[4]. Typically, a data engineer analyses these test monitoring data to extract relevant information about the neural network's recognition. These extracted information is then specified as the key-properties with the SEMKIS-DSL.

[4]Many statistical results about the neural network's recognition skills.

In this section, we propose the concept of key-properties supported by the SEMKIS-DSL to specify the neural network's recognition skills. Key-properties serve globally to describe the learned skills of target neural network acquired during the training. In summary, these key-properties are required to achieve the following goals:

- Understanding the recognition skills of a trained neural network.
- Comparing them with the requirements in order to validate and evaluate a trained neural network.
- Defining precise data needs for reengineering the training, testing and development dataset augmentation.

Key-properties can be grouped in two main categories, the quantitative and qualitative key-properties. Thus, we defined two different concepts for these two key-property categories:

- **Quantitative key-properties (ctQuantitativeKeyproperties)** are attributes describing the recognition skills that are represented as a numerical value. (e.g. dataset size, accuracy, loss, number of recognised data,. . .)
- **Qualitative key-properties (ctQualitativeKeyproperties)** are attributes describing the recognition skills that are represented as a textual or boolean expression. (e.g. features, data content,. . .)

In these key-property categories, there exist many types of key-properties. We have identified several subcategories of key-properties, which we present in the next part of this section. Quantitative key-properties may be categorised into the subcategory of continuous or discrete key-properties. Qualitative key-properties may be categorised into the subcategory of logical, nominal or ordinal key-properties.

- **Continuous key-properties** are quantitative key-properties having their value belonging to a non-countable set. Typically, these key-properties describe the accuracy, loss, precision and recall of the trained neural network resulting from the recognition of training, development and test dataset. Their value is usually represented as a real number. Examples of such key-properties would be the reached loss (**ctReachedLoss**), accuracy (**ctReachedAccuracy**), precision (**ctReachedPrecision**) and recall (**ctRecall**).
- **Discrete key-properties** are quantitative key-properties having their value belonging to a countable set. Typically, these key-properties describe some statistics on the test monitoring data and the datasets. Among others, we identified these instances of useful key-properties:

  - Size of the datasets
  - Number of data per equivalence class
  - Number of correctly classified and correctly recognised data
  - Number of correctly classified and incorrectly recognised data
  - Number of incorrectly classified and incorrectly recognised data
  - Number of incorrectly classified and incorrectly recognised data

- **Logical key-properties** are qualitative key-properties. These key-properties describe any boolean attributes resulting from the test monitoring data. These key-properties shall be described numerically, but shall describe conclusions made on the recognition of the datasets, the equivalence classes and the data itself. Examples are the correctness of the classification of all the data or the correctness of the recognition of all the data (**ctEquivClassRecoCorrectness** and **ctDataClassRecoCorrectness**). Of course, the correctness can be defined precisely for every data element or globally looking at an equivalence class.

- **Nominal key-properties** are qualitative key-properties. These key-properties are usually represented as a text. They describe any characteristics on the recognition skills of the neural network as well as testing issues. Additionally, they serve for describing detected anomalies and strong recognitions of the neural network. Examples are any recognition anomaly in an equivalence class (**ctRecoAnomaly**), not sufficiently tested equivalence classes (**ctUntestedEClasses**) or any recognition characteristics of the data belonging to an equivalence class (**ctRecognitionCharacteristic**).

- **Ordinal key-properties** are qualitative key-properties. These key-properties are usually represented as a text being part of an ordered set. They serve for evaluating the recognition performance of an equivalence class and are useful to detect issues at the level of equivalence classes. Examples are the definition of the selection function defining the recognized equivalence class (**ctEquivClassSelection**) or an evaluation of the recognition of an equivalence class (**ctRecognitionEvaluation**).

The previously defined key-property categories are used to group the different key-properties. We show the key-properties specification concepts in our metamodel in Figure 4.2

Thanks to the categorisation, it is easy to extend our metamodel additional key-properties. Therefore, we have identified some key-property type for which additional key-properties can be defined and categorised into the key-property categories. Note that we only provide

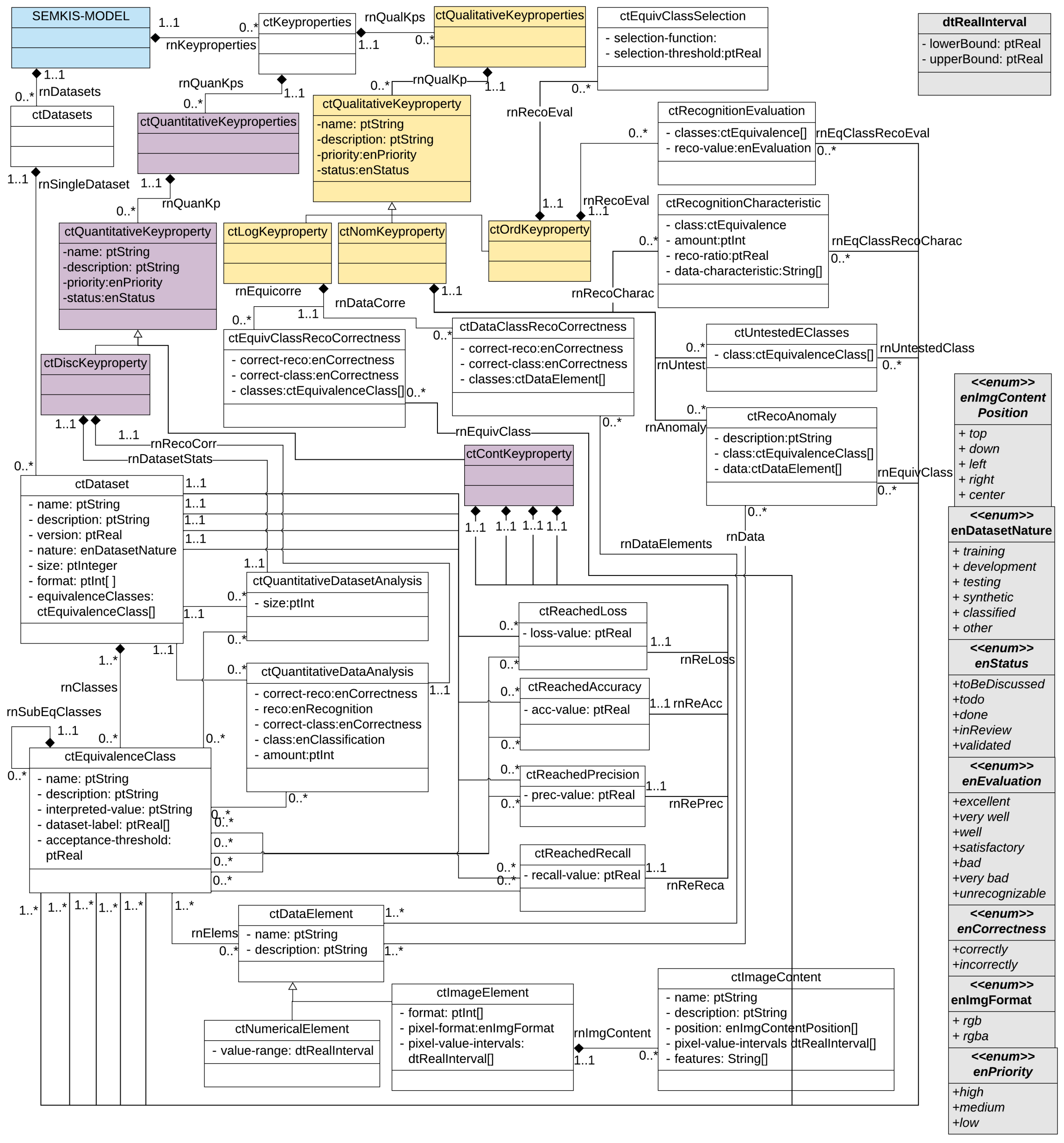


Fig. 4.2 SEMKIS-DSL Metamodel: Key-properties specification concepts.

some examples of additional key-property types for extending[5] the current version of the SEMKIS-DSL. These key-property types might be useful to describe a developed neural network version. Especially, when it comes to the presentation of a targetNN version to a customer, some additional key-properties can be useful. These key-properties allow a customer-friendly description of the neural network's recognition skills. Here are some identified key-properties types that can be caterised into our key-property categories:

- **Neural network's execution** describing any parameters related to the neural network's execution (e.g. execution time, limitations, hardware needs)
- **Neural network's reliability** describes whether the neural network tends to over- or underfit. (e.g. any issues concerning the recognition of data.)
- **Neural network's availability** describes whether the system is operational at some time. (e.g. average timeouts, probability of systems failure, ...)
- **Dataset Evaluation** describes any detected issues related to the dataset (e.g. insufficient data, definition of the equivalence classes, correctly data selection, ...)
- **Equivalence Class Evaluation** describes any detected issues at the level of equivalence classes (e.g. sufficient amount of data, data variations, data bias, ...)
- **Testing evaluation** describes if the neural network has been sufficiently tested (e.g. amount of data, testing time, testing strategy, requirements test coverage, ...)

In the upcoming sections, we will present the concrete syntax of our domain-specific language allowing to specify these presented concepts.

4.3 Concrete syntax

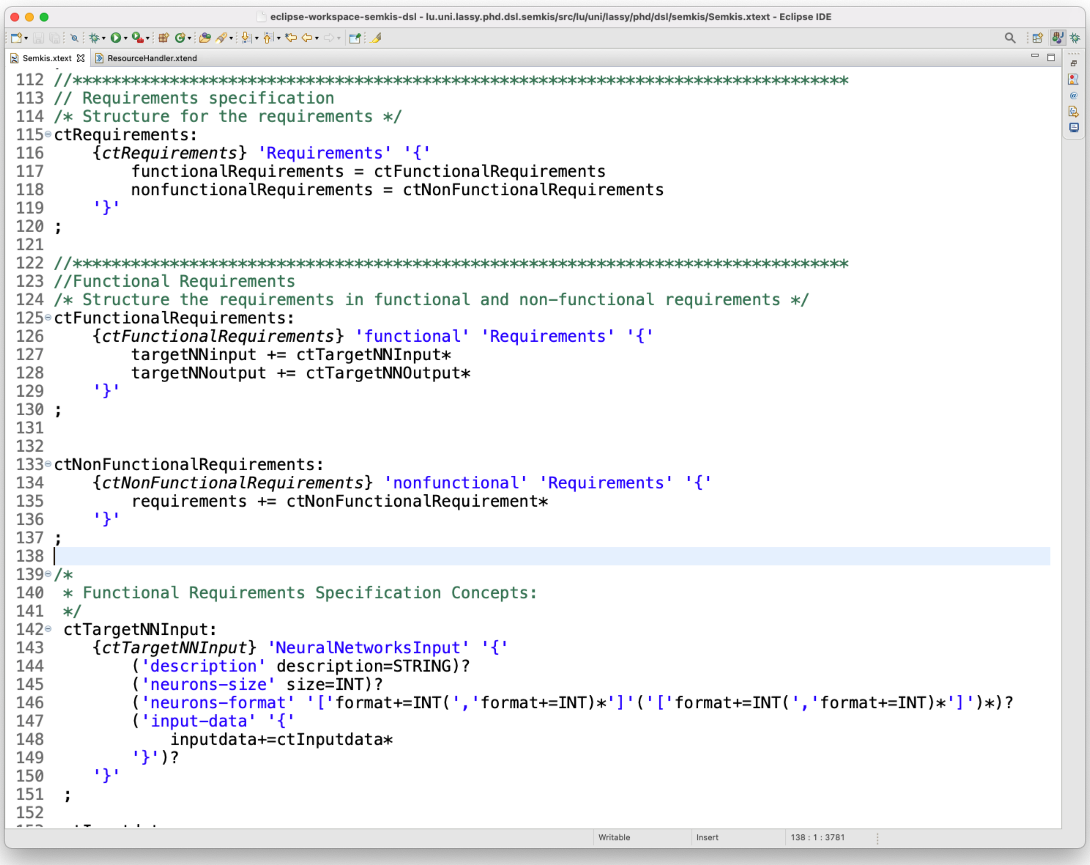
In this section, we present the concrete syntax of each SEMKIS-DSL concept introduced in the previous section.

The SEMKIS-DSL has been developed using the Xtext framework. Xtext [123] is an open-source framework for engineering textual domain-specific languages and building textual editors. We present our grammar model and illustrate the grammar with our running example from the previous chapter of section 3.2. In that running example, we presented an instance of an engineering process of a neural network capable of recognizing handwritten

[5]In this version of the SEMKIS-DSL, we defined only the most important concepts required for specifying the requirements and key-properties.

digits for a customer. In the previous section, we illustrated some manual specifications of requirements and key-properties. We reuse the specified notions of the previous section to present some specification examples with the SEMKIS-DSL.

Figure 4.3 shows the Xtext environment with some of our grammar rules of the SEMKIS-DSL.


eclipse-workspace-semkis-dsl - lu.uni.lassy.phd.dsl.semkis/src/lu/uni/lassy/phd/dsl/semkis/Semkis.xtext - Eclipse IDE
Semkis.xtext
ResourceHandler.xtend
Writable
Insert
138 : 1 : 3781

```
//*****************************************************************************
// Requirements specification
/* Structure for the requirements */
ctRequirements:
    {ctRequirements} 'Requirements' '{'
        functionalRequirements = ctFunctionalRequirements
        nonfunctionalRequirements = ctNonFunctionalRequirements
    '}'
;

//*****************************************************************************
//Functional Requirements
/* Structure the requirements in functional and non-functional requirements */
ctFunctionalRequirements:
    {ctFunctionalRequirements} 'functional' 'Requirements' '{'
        targetNNinput += ctTargetNNInput*
        targetNNoutput += ctTargetNNOutput*
    '}'
;


ctNonFunctionalRequirements:
    {ctNonFunctionalRequirements} 'nonfunctional' 'Requirements' '{'
        requirements += ctNonFunctionalRequirement*
    '}'
;

/*
 * Functional Requirements Specification Concepts:
 */
 ctTargetNNInput:
    {ctTargetNNInput} 'NeuralNetworksInput' '{'
        ('description' description=STRING)?
        ('neurons-size' size=INT)?
        ('neurons-format' '['format+=INT(','format+=INT)*']'('['format+=INT(','format+=INT)*']')*)?
        ('input-data' '{'
            inputdata+=ctInputdata*
        '}')?
    '}'
 ;

```

Fig. 4.3 Grammar of our SEMKIS-DSL in Eclipse with the Xtext Framework.

Our concrete syntax has been designed to be intuitive, we have selected mainly keywords coming from the software engineering concepts to facilitate the usage of the DSL for software engineers. The language does not use any logical operators and control flow statements. Depending on the recognition problems, some concepts has to be added for to the grammar in order to specify them. We use cross-references and containment relation to improve the structuring of specifications and to ease the access of specified information.

4.3.1 Datasets, Equivalence Classes and Data Elements Specification

4.3.1.1 Grammar rules

In this section, we present the first grammar rules related to the requirements and the key-properties. These concepts are typically specified during the requirements engineering phase. In Listing 4.1, we illustrate the grammar rules related to the requirements.

The software engineer specifies the datasets used for training and testing the target neural network. Typically, the engineer specifies multiple dataset with the SEMKIS-DSL. In order to specify the datasets, the engineer starts by defining a set of dataset specifications (**ctDatasets**). This set contains all specifications of the different datasets (**ctDataset**).

Each dataset specification is defined with the following attributes :

- **name** describes the name of the dataset.
- **description** is a textual field describing the dataset.
- **dataset-type** describes whether we specify a testing, training or development dataset.
- **version** describes a particular instance of a dataset to show its evolution.
- **size** describes the number of elements in the dataset.
- **format** describes the shape of the stored information inside a dataset.
- **equivalence classes** is the set of equivalence classes represented in this dataset.

The next concept that can be specified with the SEMKIS-DSL are the equivalence classes (**ctEquivalenceClasses**). In supervised learning, equivalence classes represent groups of data that are part of a dataset. Typically, the targetNN shall recognize the equivalence class to which a data element belongs. Equivalence classes (**ctEquivalenceClasses**) are specified in a single block of equivalence class specifications(**ctEquivalenceClass**).

Each equivalence class specification is defined with the following attributes:

1. **name** represents the name of the equivalence class.
2. **description** represents a textual expression describing the equivalence class.
3. **value** represents the actual value of the equivalence class. It defines what the target neural network shall recognize.
4. **acceptance-rule** defines a rule accepting that a data element belongs to the equivalence class.

5. **data-elements** is a list of data elements that shall be recognized in this equivalence class.
6. **subclasses** are potential subdivisions of equivalence class. This is needed, if it is required to divide data element of one equivalence class into multiple equivalence classes.

```
ctDatasets :
    {ctDatasets} 'Datasets' '{'
        datasets += ctDataset*
    '}';

    ctDataset:
        'Dataset' name=ID ('for' 'process-iteration' processiteration=INT)? '{'
            ('name' fullname=STRING)
            ('description' description=STRING)?
            ('dataset-type' type=enDATASETTYPE)?
            ('version' version=REAL)?
            ('size' size=INT)?
            ('format' '['format+=INT('x'format+=INT)*']')?
            ('equivalenceclasses' '{'
                equivalenceclasses+=[ctEquivalenceClass|FQN]('(''numElements'
                    classsize+=INT')')?(','equivalenceclasses+=[ctEquivalenceClass|
                    FQN] ('(''numElements' classsize+=INT')')?)*
                '}'
            )?
        '}' ;

    ctEquivalenceClasses :
        {ctEquivalenceClasses} 'EquivalenceClasses' '{'
            equivalenceClasses += ctEquivalenceClass*
        '}';

    ctEquivalenceClass:
        'EquivalenceClass'name=ID '{'
            ('name' fullname=STRING)
            ('description' description=STRING)?
            ('value' value=STRING)?
            ('acceptance-rule' symbol=enCOMPARATORS threshold=REAL)?
            ('data-elements' dataelements+=[ctDataElement|FQN](','dataelements+=[ct
                DataElement|FQN])*)?
            rnSubEquivalenceClasses+=ctEquivalenceClass*
        '}';
```

Listing 4.1 SEMKIS-DSL grammar rules: Dataset, Equivalence Classes and Data Elements

A dataset consists of labelled data elements. A label is represented as equivalence class (e.g. images of a handwritten digit labelled 1 is in the equivalence class 1). A shape of a data element might be a single value (e.g. age, year, percentage, nationality) or a one/multi-dimensional vector (e.g. image, voice, ...) The data elements (**ctDataElements**) are specified in a block of data element specifications (**ctDataElement**). Listing 4.2 illustrates the grammar rules associated to the data elements. A data element (**ctDataElement**) is defined with the following attributes:

- **name** is a textual field for the name of the data element. (e.g. ImagesOfDigitOne)
- **description** is a textual field describing these data elements.
- **numerical or image elements** are any types of data elements that can occur in a dataset. The software engineer selects a type based on the input requirements of the customer.

A data element might be a numerical or an image element in the SEMKIS-DSL. The SEMKIS-DSL currently supports only the specification of these two types of elements. However, the DSL can be extended if is required to specify other data elements. In this thesis, we focus mainly on image recognition, which is the reason, why we included this specification.

A data element may be a numerical element (e.g. age, year, percentage. . .). It is defined with these attributes:

- **name** is a textual field for the name of the numerical element. (e.g. age, year,...)
- **description** is a textual field describing these numerical elements.
- **value-range** is an interval in which the numerical value is defined.

A data element may also be an image element (e.g. photos, images, drawings, handwritten text,...) It is defined with these attributes:

- **name** is a textual field for the name of the image element. (e.g. images, photos, drawings, ...)
- **description** is a textual field describing these image elements.
- **format** defines the actual image sizes, typically any tensor size.
- **pixel-format** defines the type of format used for representing the picture (e.g. rgb, rgba,. . .)
- **pixel-value-intervals** defines the minimal and maximal value of a pixel.
- **image-content** defines the objects represented on an image.

An image consists of some image content that can also be specified with the SEMKIS-DSL. We defined a grammar rule for the image content (**ctImageContent**), which has the following attributes:

- **name** is a textual field for the name of the image content. (e.g. a car, a handwritten digit, a letter, ...)
- **description** is a textual field describing these image object.
- **position** defines the place of the object in the image.
- **pixel-value-intervals** defines the minimal and maximal pixel value of the object.
- **features** is a list of words to characterise these objects.

Note that we describe here mainly images constructed as 3D-matrix of pixels values in $[0,255]$, where each layer represents the colors red, green and blue of the RGB-format. However, it is still possible to specify images having another format by including a grammar rule and adding it to the DataElement. The grammar remains maintainable in case additional features are required.

```
ctDataElements :
    {ctDataElements} 'DataElements' '{'
        dataElement += ctDataElement*
    '}';

ctDataElement:
    'DataElement' name=ID '{'
        ('name' fullname=STRING)?
        ('description' description=STRING)?
        (('numerical-elements' '{'
            numericalelements+=ctNumericalElement*
        '}'
        )|
        ('image-elements' '{'
            imageobjects+=ctImageElement*
        '}'
        ))?
    '}';

ctNumericalElement:
    'NumericalElement' name=ID '{'
        ('name' fullname=STRING)?
        ('description' description=STRING)?
        ('value-range' interval=dtRealInterval)?
    '}';

ctImageElement:
    'ImageElement' name=ID '{'
        ('name' fullname=STRING)?
        ('description' description=STRING)?
        ('format' '['format+=INT('x'format+=INT)*']')?
        ('pixel-format' pixelformat=enPIXELFORMAT)?
        ('pixel-value-intervals' intervals+=dtRealInterval(';'intervals+=
            dtRealInterval)*)?
        ('image-content' '{'
            imagecontent+=ctImageContent*
        '}'
        )?
    '}';

ctImageContent:
    'ImageContent' name=ID '{'
        ('name' fullname=STRING)?
        ('description' description=STRING)?
        ('position' enImgContentPosition=enPOSITION)?
        ('pixel-value-intervals' intervals+=dtRealInterval(';'intervals+=
            dtRealInterval)*)?
        ('features' features+=STRING(','features+=STRING)*)?
    '}';
```

Listing 4.2 Data elements

4.3.1.2 Running Example - Dataset, Equivalence Class and Data Element specification

In the context of our running example, we specify the datasets and equivalence classes required for recognising the MNIST handwritten digits. This task is typically performed in the pre-activity of the SEMKIS process presented in section 3.6.1. We won't consider the entire MNIST recognition problem, but we focus only on the recognition of the handwritten digits 'zero' and 'seven'. This is the reason, why we present only a partial specification of the requirements and key-properties. Note that we use the running example to show a concrete specification on a basic problem. In Listing 4.3, we show the partial specification of the datasets, equivalence classes and data elements.

```
    Datasets {
        Dataset dstest for process-iteration 0 {
            name 'testing dataset'
            description 'Dataset used for testing the targetNN after training'
            dataset-type testing
            version 0.1
            size 2008
            format [28 x 28 x 2008] //describes the entire dimensions of the dataset
            equivalenceclasses{
                zero    (numElements 980),
                seven   (numElements 1028)
            }}}
    EquivalenceClasses {
        EquivalenceClass zero {
            name 'Digit-One-Images'
            description 'Contains all classified and recognised images of a
                handwritten 0'
            value '0'
            acceptance-rule greaterThan 0.9
            data-elements imgDigitZero
        }
        EquivalenceClass seven {
            name 'Digit-Seven-Images'
            description 'Contains all classified and recognised images of a
                handwritten 7'
            value '7'
            acceptance-rule greaterThan 0.9
            data-elements imgDigitSeven
        }}
    DataElements {
        DataElement imgDigitZero {
            name 'Digit Zero'
            description 'Image of a white 0 on  black background'
            image-elements {
                ImageElement img0{
                    format [28 x 28]
                    pixel-format rgb
                    pixel-value-intervals [0;255]
                    image-content {
                        ImageContent img0digit {
                            name 'Digit Zero'
                            description 'White handwritten digit 0'
                            position center
                            pixel-value-intervals [220;255]
                            features 'white shape'
                        }
                        ImageContent img0bg {
                            name 'Background'
                            description 'Black background'
                            position background
                            pixel-value-intervals [0;50]
                            features 'black'
                        }
                    }}}}
        DataElement imgDigitSeven {
            name 'Digit Seven'
            description 'Image of a white 7 on  black background'
            image-elements {
                ImageElement img7{
                    format [28 x 28]
                    pixel-format rgb
                    pixel-value-intervals [0;255]
                    image-content {
                        ImageContent img7digit {
                            name 'Digit Seven'
                            description 'White handwritten digit 7'
                            position center
                            pixel-value-intervals [220;255]
                            features 'white shape', 'middleDash', 'curved','
                                noMiddleDash'
                        }
                        ImageContent img7bg {
                            name 'Background'
                            description 'Black background'
                            position background
                            pixel-value-intervals [0;50]
                            features 'black'
                        }}}}}
    }
```

Listing 4.3 Dataset and equivalence class specification

4.3.2 Requirements specification

In this section, we present the grammar rules needed for specifying the customer's requirements [6]. In addition to that, we illustrate our language with some specification examples of requirements in the context of our running example.

4.3.2.1 Grammar rules

The first step towards a neural network is the specification of the customer's requirements. Requirements serve to describe the customer's needs for a neural network. They are used as input of the SEMKIS process to engineer the datasets and the target neural network. Therefore, we want to support software engineering during their requirements specification tasks with our SEMKIS-DSL to engineer improved neural network's that satisfy the customer's requirements.

The first grammar rule is the concept of requirements (**ctRequirements**). This rule is the starting point of a specification and contain all functional (**ctFunctionalRequirements**) and nonfunctional requirements (**ctNonFunctionalRequirements**). Therefore, we defined two grammar rules for the different types of requirements, the functional and nonfunctional requirements. The requirements are associated with a containment relation to functional and non-functional requirements.

Functional requirements (**ctFunctionalRequirements**) contain a specification of the targetNN's input and the targetNN's output. We defined two grammar rules, the targetNN's in-

[6]The customer describes his needs of a neural network to a software engineering.

put (**ctTargetNNInput**) and output (**ctTargetNNOutput**) associated with a containment relation to functional requirements.

Nonfunctional requirements (**ctNonFunctionalRequirements**) consist of a specification of quantitative properties describing the targetNN's recognition. These quantitative recognition properties might be the accuracy, the loss, the recall or precision. We defined distinct grammar rules for of these properties. They are associated with a containment relation to the nonfunctional requirements. In Listing 4.4, we show the presented grammar rules of the SEMKIS-DSL.

```
ctRequirements:
    {ctRequirements} 'Requirements' '{'
        functionalRequirements = ctFunctionalRequirements
        nonfunctionalRequirements = ctNonFunctionalRequirements
    '}';

ctFunctionalRequirements:
    {ctFunctionalRequirements} 'functional' 'Requirements' '{'
        targetNNinput += ctTargetNNInput*
        targetNNoutput += ctTargetNNOutput*
    '}';

ctNonFunctionalRequirements:
    {ctNonFunctionalRequirements} 'nonfunctional' 'Requirements' '{'
        accuracy = ctTargetAccuracy
        loss = ctTargetLoss
        recall = ctTargetRecall
        precision = ctTargetPrecision
    '}';
```

Listing 4.4 Functional and Nonfunctional requirements

The next grammar rules presented in this section are focusing on describing the different types of functional requirements. We have a grammar rules for the targetNN's input (**ctTargetNNInput**), the targetNN's output (**ctTargetNNOutput**). Listing 4.5 contains all grammar rules related to the functional requirements. The targetNN's input has the following attributes:

- **neurons-size** defines the number of neurons at the input of the target neural network. Each neuron represents some data that has to be processed through the network. It is represented as an integer value.
- **neurons-format** defines the input format of the target neural network's input data. e.g. vector format, matrix format or any tensor format,...

- **input-data** is a list of input data (**ctInputData**), where the engineer may specify any kind of data serving as input for the target neural network. For each input data, we have defined:
  - **neurons-values** defines the value range of some input data. The attribute describes the possible input values of an input neuron.
  - **data-elements** defines any type of data being in a dataset. This might be an image or some numerical value. It is defined by a name and a description in our grammar. Data element are typically belonging to an equivalence class e.g. Images of a handwritten digit 7 belonging to the equivalence class 7.

Finally, we defined the following attributes for the targetNN's output (**ctTargetNNOutput**):

- **neurons-size** defines the number of neurons at the output of the target neural network.
- **neurons-format** defines the output format of the target neural network's output data. e.g. vector format, matrix format or any tensor format,...
- **output-value-range** defines a real value interval to which the output values belong.
- **output-neurons** defines a list of equivalence class. The target neural network recognizes some data element and outputs some information through each output neuron. Thus, it is necessary to define the equivalence classes representing each output neuron.

```
ctTargetNNInput:
    {ctTargetNNInput} 'NeuralNetworksInput' '{'
        ('neurons-size' size=INT)?
        ('neurons-format' '['format+=INT(','format+=INT)*']'('['format+=INT(','
            format+=INT)*']')*)?
        ('input-data' '{'
            inputdata+=ctInputdata*
        '}')?
    '}';

ctInputdata:
    'InputData' name=ID '{'
        ('neuron-values' interval=dtRealInterval)
        ('data-elements' dataelements+=[ctDataElement|FQN](','dataelements+=[ct
            DataElement|FQN])*)?
    '}';

ctTargetNNOutput:
    {ctTargetNNOutput} 'NeuralNetworksOutput' '{'
```

```
            ('neurons-size' size=INT)?
            ('neurons-format' '['format+=INT(','format+=INT)*']''('['format+=INT(','
                format+=INT)*']')*)?
            ('output-value-range' interval=dtRealInterval)?
            ('output-neurons' '['equivalenceclass+=[ctEquivalenceClass|FQN](','
                equivalenceclass+=[ctEquivalenceClass|FQN])*']')?
        '}'
    ;
```

Listing 4.5 Functional requirements

Lastly, we present the grammar rules for the different types of nonfunctional requirements. We defined for the four different concepts; target accuracy, target loss, target precision and target recall; four grammar rules that can be specified with the SEMKIS-DSL. Listing 4.6 contains all grammar rules related to the nonfunctional requirements. These four performance concepts follow a similar format, which we present here:

- **description** is a textual field describing a performance concept
- **purpose** defines the actual need for this performance concept.
- **priority** defines the necessity of the satisfaction of this requirements.
- **equivalenceclasses** defines on which equivalence class the performance parameter shall be satisfied. Note that if this field is empty, it is meant to be the overall performance on the dataset.
- **min-XYZ** defines the minimal accepted value of the performance concept, where XYZ stands for one of them.
- **max-XYZ** defines the maximal accepted value of the performance concept, where XYZ stands for one of them.
- **target-XYZ** defines the target accepted value of the performance concept, where XYZ stands for one of them.

```
    ctTargetAccuracy:
        'TargetAccuracy' name=ID ('for' 'dataset' dataset=[ctDataset|FQN])? '{'
            ('description' description=STRING)?
            ('purpose' purpose=STRING)?
            ('priority' priority=PRIORITY)?
            ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
                equivalenceclass+=[ctEquivalenceClass|FQN])*)?
            ('min-accuracy' min_acc=REAL)?
            ('max-accuracy' max_acc=REAL)?
            ('target-accuracy' target_acc=REAL)?
        '}'
    ;

    ctTargetLoss:
        'TargetLoss' name=ID ('for' 'dataset' dataset=[ctDataset|FQN])? '{'
            ('description' description=STRING)?
            ('purpose' purpose=STRING)?
            ('priority' priority=PRIORITY)?
            ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
                equivalenceclass+=[ctEquivalenceClass|FQN])*)?
            ('min-loss' min_loss=REAL)?
            ('max-loss' max_loss=REAL)?
            ('target-loss' target_loss=REAL)?
        '}'
    ;

    ctTargetRecall:
        'TargetRecall' name=ID ('for' 'dataset' dataset=[ctDataset|FQN])? '{'
            ('description' description=STRING)?
            ('purpose' purpose=STRING)?
            ('priority' priority=PRIORITY)?
            ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
                equivalenceclass+=[ctEquivalenceClass|FQN])*)?
            ('min-recall' min_recall=REAL)?
            ('max-recall' max_recall=REAL)?
            ('target-recall' target_recall=REAL)?
        '}'
    ;

    ctTargetPrecision:
        'TargetPrecision' name=ID ('for' 'dataset' dataset=[ctDataset|FQN])? '{'
            ('description' description=STRING)?
            ('purpose' purpose=STRING)?
            ('priority' priority=PRIORITY)?
            ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
                equivalenceclass+=[ctEquivalenceClass|FQN])*)?
            ('min-precision' min_precision=REAL)?
            ('max-precision' max_precision=REAL)?
            ('target-precision' target_precision=REAL)?
        '}'
    ;
```

Listing 4.6 Nonfunctional requirements

4.3.2.2 Running Example - Requirements specification

The next step in our running example is the specification of the requirements. This task is typically performed in the pre-activity of the SEMKIS process presented in section 3.6.1. In this section, we specify basically some functional and nonfunctional requirements to describe the target neural network.

We do not specify all the requirements, but we present here a snapshot of a potential specification to illustrate the usage of the SEMKIS-DSL. In Listing 4.3, we show a specification of the requirements for the MNIST problem. The specification contains some functional and nonfunctional requirements. The functional requirements describe the neural network's input and targeted output and the non-functional requirements describe some targeted recognition values. These values describe the accuracy on the whole testing dataset and the equivalence class 7.

```
Requirements {
    functional Requirements {
        NeuralNetworksInput {
            neurons-size 784 //input size of the neural network (28 times 28)
            neurons-format [28,28]
            input-data {
                InputData iptSeven{
                    neuron-values [0;255]
                    data-elements imgDigitSeven
                }
            }

        }

        NeuralNetworksOutput {
            neurons-size 10
            neurons-format [1,10]
            output-value-range [0;1]
            output-neurons [seven]
            input-data iptSeven
        }
    }
    nonfunctional Requirements {
        Requirement NFR3 {
            description 'Global accuracy on testing dstest'
            purpose 'Defining the tolerated accuracy on the testing dataset'
            priority high
            TargetAccuracy tarGlobalTestAcc for dataset dstest{
                min-accuracy 98
                max-accuracy 100
                target-accuracy 99.7
            }
        }

        Requirement NFR7 {
            description 'Accuracy on testing dataset for ec7'
            purpose 'Defining the tolerated accuracy for ec7 on the testing data
                set'
            priority high
            TargetAccuracy tarEcSevenAcc for dataset dstrain{
                equivalenceclasses seven
                min-accuracy 99
```

```
                        max-accuracy 100
                        target-accuracy 99.9
                    }
                }
            }
        }
```

Listing 4.7 Requirements specification

4.3.3 Key-Properties specification

In this section, we present the grammar rules required for the specification of the key-properties (see section 3.6.7.2 definition 5). We illustrate some grammar rules with some specification examples of key-properties in the context of our running example.

4.3.3.1 Grammar Rules

According to the SEMKIS process, the software engineer analyses the test monitoring data (see section 3.6.6) immediately after the testing of the target neural network. During this activity, the software engineer learns more about the target neural network's recognition skills. In addition, the acquired knowledge of the recognition skills shall be specified using the SEMKIS-DSL as 'key-properties'. The specification of the key-properties has several advantages:

- Allows verifying whether the requirements have been satisfied
- Serves as a basis to generate a dataset augmentation specification in order to reeningeer the datasets to improve the target neural network

Key-properties are specified in a single block, containing the entire key-properties specification (**ctKeyproperties**). They can be specified as qualitative or as quantitative key-properties. Listing 4.8 shows the grammar rules for the key-property specification.

Qualitative key-properties (**ctQualitativeKeyProperties**) are describing recognition skills that are specified textually or with boolean expressions. For example, a software engineer specifies these key-properties to describe precisely the recognition of specific objects in images, entire equivalence classes, any recognition anomlies,... Qualitative key-properties can be specified in as nominal, ordinal or logical key-properties. We present the three types of qualitative key-properties in the list below:

- Nominal key-properties (**ctNominalKeyProperty**) are textually specified properties to describe the recognition of specific data or content of data. The software engineer specifies untested classes and recognition anomalies with these properties.

- Ordinal key-properties (**ctOrdinalKeyProperty**) are textually specified properties that are part of an ordered set. Software engineers specify the evaluation of the recognition of equivalence classes.
- Logical key-properties (**ctLogicalKeyProperty**) are specified boolean-based properties serving to describe which data and equivalence classes have been correctly recognized and classified.

Quantitative key-properties (**ctQuantitativeKeyProperties**) are describing recognition skills that are specified with numerical values (e.g. dataset size, accuracy, loss, recall, precision,...) For example, a software engineer specifies quantitative key-properties to describe statistics on the recognition of the test data,... Quantitative key-properties can be specified in as continuous or discrete key-properties. We present the two types of quantitative key-properties in the list below:

- Continuous key-properties (**ctContinuousKeyProperty**) are numerical values being part of a non-countable set. The software engineer specifies values such as accuracy, loss, precision and recall.
- Discrete key-properties (**ctDiscreteKeyProperty**) have numerical values being part of a countable set. Software engineers specify dataset size, categorisations, number of correctly and incorrectly classified data...

```
ctKeyProperties:
    {ctKeyProperties} 'KeyProperties' '{'
        qualKeyproperties += ctQualitativeKeyProperties*
        quanKeyproperties += ctQuantitativeKeyProperties*
    '}';

ctQualitativeKeyProperties:
    {ctQualitativeKeyProperties} 'QualitativeKeyProperties' '{'
        nominalkeyproperties += ctNominalKeyProperty*
        ordinalkeyproperties += ctOrdinalKeyProperty*
        logicalkeyproperties += ctLogicalKeyProperty*
    '}';

ctQuantitativeKeyProperties:
{ctQuantitativeKeyProperties} 'QuantitativeKeyProperties' '{'
    continuouskeyproperties += ctContinuousKeyProperty*
    discretekeyproperties += ctDiscreteKeyProperty*
'}';
```

Listing 4.8 KeyProperties Specification

The first qualitative key-property that we present in this section are nominal key-properties (**ctKeyProperties**). Nominal key-properties serve to describe the recognition characteristics, define the untested equivalence classes and specify any recognition anomalies. Thus, software engineer can use the SEMKIS-DSL to describe these concepts. Listing 4.9 illustrates the grammar rules related to the nominal key-property specification.

The block of nominal key-properties contains the specification of the untested equivalence classes (**ctUntestedClasses**), recognition characteristics (**ctRecognitionCharacteristic**) and the recognition anomalies (**ctRecoAnomaly**).

The first grammar rules of the nominal key-properties, which we present, are the untested equivalence classes (**ctUntestedClasses**). These rules allow the specification of the equivalence classes that have not been at all or sufficiently tested. Untested classes are defined with two attributes, a textual description and the equivalence classes, that are required to be tested.

Recognition anomalies are specified for describing any concrete issues on the recognition of data elements or entire equivalence classes. For example, the software engineer might specify any problems with the recognition of objects on images. Recognition anomalies are specified with these attributes:

- **description** is a textual value describing the recognition anomaly.
- **data-elements** is a list containing all data elements where recognition anomalies have been detected.
- **equivalence classes** is a list containing all equivalence classes where recognition anomalies have been detected.

Recognition characteristics are specified for describing concrete recognition skills of the targetNN. The specification can contain precise descriptions of that targetNN's recognition of the data-elements. Software engineers may describe precisely detected correct recognitions of the targetNN.

- **characteristics** is a list of one-word textual descriptions of the targetNN's recognitions. These characteristics describe concretely what the targetNN is able to recognise.
- **equivalenceclasses** is a list containing the equivalence classes from which the characteristics have been detected.
- **recognizedAmount** is the amount of data-elements from which the characteristics have been detected.
- **recognition-ration** is the ratio of data-elements from which the characteristics have been detected.

```
ctNominalKeyProperty:
    'NominalKeyProperty' name=ID '{'
        untestedClasses += ctUntestedClasses*
        recognitioncharacteristics += ctRecognitionCharacteristic*
        recoAnamalies += ctRecoAnomaly*
    '}'
;

ctRecognitionCharacteristic:
    'RecognitionCharacteristic' name=ID '{'
        ('recognizedAmount' amount=INT)?
        ('recognition-ration' ratio=REAL)?
        ('characteristics' characteristics+=STRING(','characteristics+=STRING*))?
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?
    '}'
;
ctUntestedClasses:
    'UntestedClasses' name=ID '{'
        ('description' description=STRING)?
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?
    '}'
;
ctRecoAnomaly:
    'RecognitionAnomaly' name=ID '{'
        ('description' description=STRING)?
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?
        ('data-elements' dataelements+=[ctDataElement|FQN](','dataelements+=[ct
            DataElement|FQN])*)?
    '}'
;
```

Listing 4.9 Nominal key-properties Specification

The block of ordinal key-properties contains the specification of recognition evaluation and the selection criteria for a data-elements belonging to an equivalence class. Listing 4.10 shows the grammar rules for the specification of the ordinal key-properties.

The first grammar rules defines the specification of the recognition evaluation. Software engineers can evaluate the recognition of certain equivalence classes. The recognition evaluation (**ctRecognitionEvaluation**) is defined with these attributes:

- **equivalenceclasses** is a list of equivalence classes that the engineer wants to evaluate.
- **recognition-value** is an evaluation criterion from excellent to unrecognizable. The software engineer may define whether the equivalence class has been recognised well.

Another grammar rules defines the selection of the equivalence class for a recognised data-element. The software engineer can specify the selection criteria of the targetNN's output in order to recognise a processed data-element in a certain equivalence class. The equivalence class selection (**ctEquivalenceclassSelection**) is defined with these attributes:

- **selection-function** is a method used for selecting the targetNN's output and associating it to an equivalence class.
- **selection-threshold** is a threshold for defining, which values are important to be analysed for recognising the equivalence class.

modelRefStyle

```
ctOrdinalKeyProperty:
    'OrdinalKeyProperty' name=ID '{'
        recognitionevaluations += ctRecognitionEvaluation*
        equivalenceclassselection += ctEquivalenceclassSelection*
    '}'
;

ctRecognitionEvaluation:
    'RecognitionEvaluation' name=ID '{'
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?
        ('recognition-value' recognitionvalue=enEVALUATION)?
    '}'
;
ctEquivalenceclassSelection:
    'EquivalenceclassSelection' name=ID '{'
        ('selection-function' selection=enSELECTION)?
        ('selection-threshold'  symbol=enCOMPARATORS threshold=REAL)?
    '}'
;
```

Listing 4.10 Ordinal KeyProperties Specification

The block of logical key-properties contains the specification of the recognition correctness of equivalence classes and data-elements. Listing 4.11 shows the grammar rules for the specification of the logical key-properties.

The first grammar rule defines the specification of the recognition correctness of an entire equivalence class. The software engineer specifies his overall evaluation of the equivalence class, whether the data-elements have been correctly recognised and correctly classified. The recognition correctness of equivalence classes (**ctEquivClassRecoCorrectness**) is defined with these attributes:

- **recognition-correctness** defines whether most data-elements in an equivalence class have been correctly recognised (defined in section 3.6.7.2).

- **classification-correctness** defines whether most data-elements in an equivalence class have been correctly classified (defined in section 3.6.7.2).
- **equivalenceclasses** is a list of equivalence classes, which the software engineer wants to evaluate.

The second grammar rule defines the specification of the recognition correctness of individual data-elements. The software engineer specifies whether individual data-elements have been correctly recognised and correctly classified. Similar to the previous grammar rules, the recognition correctness of data elements (**ctDataClassRecoCorrectness**) is defined with these attributes:

- **recognition-correctness** defines whether the data-elements have been correctly recognised (defined in section 3.6.7.2).
- **classification-correctness** defines whether the data-elements have been correctly classified (defined in section 3.6.7.2).
- **equivalenceclasses** is a list of equivalence classes, which the software engineer wants to evaluate.

```
ctLogicalKeyProperty:
    'LogicalKeyProperty' name=ID '{'
        ecRecognitioncorrectnesses += ctEquivClassRecoCorrectness*
        dataRecognitionCorrectnesses += ctDataClassRecoCorrectness*
    '}';

ctEquivClassRecoCorrectness:
    'EquivalenceClassRecognitionCorrectness' name=ID '{'
        ('recognition-correctness' recognitioncorrectness = enCORRECTNESS)?
        ('classification-correctness' classificationcorrectness = enCORRECTNESS)?
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?
    '}';

ctDataClassRecoCorrectness:
    'DataElementRecognitionCorrectness' name=ID '{'
        ('recognition-correctness' recognitioncorrectness = enCORRECTNESS)?
        ('classification-correctness' classificationcorrectness = enCORRECTNESS)?
        ('dataElements' dataelements+=[ctDataElement|FQN](','dataelements+=[ct
            DataElement|FQN])*)?
    '}';
```

Listing 4.11 Logical KeyProperties Specification

We present now the two types of quantitative key-properties. A block of quantitative key-properties may contain multiple specifications of continuous key-properties or discrete key-properties.

The first grammar rule defines the specification of continuous key-properties. The block of continuous key-properties contains the specification of the typical AI-performance values of a targetNN. These performance values might be the accuracy, loss, precision and recall on a dataset.

A continuous key-property is defined with 6 attributes from which we have one performance attributes. This key-property is defined with the following attributes:

- **Name** defines an identity name for the key-property.
- **Description** defines a textual value describing the property.
- **Priority** defines a priority of the key-property, meaning whether is important to be considered for improving the datasets.
- **Accuracy, Loss, Precision or Recall** is a specification of the recognition performance of the targetNN.

We defined separate grammar rules for the accuracy, loss, precision and recall, which follow the same structure. These grammar rules have the following attributes:

- **equivalence-classes** is a list of equivalence classes from which the performance value has been calculated.
- **X-value** defines the actual numerical value of the performance, where X stands for accuracy, loss, precision or recall.

```
ctContinuousKeyProperty:
    'ContinuousKeyProperty' name=ID '{'
        ('name' title=STRING)?
        ('description' description=STRING)?
        ('priority' priority=PRIORITY)?
        // Reached Performance Specification Global and per equivalence class
        ((accuracy = ctReachedAccuracy)|
        (loss = ctReachedLoss)|
        (recall = ctReachedRecall)|
        (precision = ctReachedPrecision))?
    '}'
;

ctReachedAccuracy:
    'ReachedAccuracy' name=ID ('for' 'dataset' dataset=[ctDataset|FQN])? '{'
```

```
            ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
                equivalenceclass+=[ctEquivalenceClass|FQN])*)?
            ('accuracy-value' accuracy_value=REAL)?
        '}'
    ;

    ctReachedLoss:
        'ReachedLoss' name=ID ('for' 'dataset' dataset=[ctDataset|FQN])? '{'
            ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
                equivalenceclass+=[ctEquivalenceClass|FQN])*)
            ('loss-value' loss_value=REAL)?
        '}'
    ;

    ctReachedRecall:
        'ReachedRecall' name=ID ('for' 'dataset' dataset=[ctDataset|FQN])? '{'
            ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
                equivalenceclass+=[ctEquivalenceClass|FQN])*)?
            ('recall-value' recall_value=REAL)?
        '}'
    ;

    ctReachedPrecision:
        'ReachedPrecision' name=ID ('for' 'dataset' dataset=[ctDataset|FQN])? '{'
            ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
                equivalenceclass+=[ctEquivalenceClass|FQN])*)?
            ('precision-value' precision_value=REAL)?
        '}'
    ;
```

Listing 4.12 Continuous KeyProperties Specification

A block of quantitative key-properties may contain multiple specifications of continuous key-properties or discrete key-properties.

Discrete key-properties can be specified as quantitative data statistics or quantitative dataset statistics. For each of the two, discrete key-properties types, we defined two separate grammar rules that follow the same style.

Quantitative dataset statistics (**ctQuantitativeDataAnalysis**) describe the number of data-elements within an equivalence class of a dataset. The software engineer might specify the attributes defining a list of equivalence classes(equivalence-classes), the dataset(dataset) and the amount of data-elements in the dataset(amount).

Quantitative data statistics (**ctQuantitativeDatasetAnalysis**) describe the number of (in-)correctly classified or (in-)correctly recognised data-elements within equivalence classes of a dataset. The software engineer might specify the attributes defining the correctness of the classification (or recognition), the dataset, a list of equivalence classes (equivalence-classes) being part of the dataset and the amount of data-elements being correctly or incorrectly classified (or recognised).

```
ctDiscreteKeyProperty:
    'DiscreteKeyProperty' name=ID '{'
        ('name' title=STRING)?
        ('description' description=STRING)?
        ('priority' priority=PRIORITY)?
        ((quantitativedataanalysis = ctQuantitativeDataAnalysis) |
        (quantitativedatasetanalysis=ctQuantitativeDatasetAnalysis))?
    '}'
;

ctQuantitativeDatasetAnalysis:
    'QuantitativeDatasetAnalysis' name=ID ('for' 'dataset' dataset=[ctDataset|FQN
        ])?'{'
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?
        'size' size=INT
    '}'
;

ctQuantitativeDataAnalysis:
    correctness=enCORRECTNESS (recognition=enRECOGNITION |classification =
        enCLASSIFICATION) 'QuantitativeDataAnalysis' name=ID ('for' 'dataset'
        dataset=[ctDataset|FQN])?'{'
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?
        'amount' amount=INT
    '}'
;
```

Listing 4.13 Discrete KeyProperties Specification

Finally, the enumerations listed inside the different grammar rules can be found in the full grammar shown in the appendices under section A.1. We won't present the enumeration in this section as they are self-explained in the context of the grammar rules descriptions.

Nevertheless, the software engineer can use these grammar rules to various requirements and key-properties. In the next section, we present the utility of a domain-specific language in order to reengineer datasets to improve target neural networks.

4.3.3.2 Running Example - Specification of the key-properties

The next step, that we perform for our running example, is the specification of the key-properties. This task is typically performed during the activity that covers the analysis of the test monitoring data. We presented this activity of the SEMKIS process in section 3.6.7. In this section, we specify the quantitative and qualitative key-properties to describe the acquired recognition skills of our target neural network.

Note that we won't specify all possible key-properties. We only specify a subset of the key-properties to show the usage of the SEMKIS-DSL on a concrete example. In Listing 4.14, we show a specification of the quantitative key-properties in the context of our running example. We illustrate the specification of the continuous and discrete key-properties. In summary, the discrete key-properties describe some statistics on the datasets and the continuous key-properties describe the reached recognition values (accuracy, loss,...) of the target neural network.

```
KeyProperties{
QuantitativeKeyProperties {
    ContinuousKeyProperty ckp1 {
        name "test dataset accuracy"
        description "Accuracy is tolerated but does not reach the targetaccuracy
            of 99.7 percent."
        priority high
        version 1.0
        status toBeDiscussed
        ReachedAccuracy acc_test {
            accuracy-value 99.47
        }
    }

    ContinuousKeyProperty ckp2 {
        name "Digit Seven recognition accuracy"
        description "Accuracy of the digit 7 is tolerated but does not reach the
            targetaccuracy of 99.9 percent."
        priority high
        version 1.0
        status toBeDiscussed
        ReachedAccuracy acc_test_seven {
            equivalenceclasses seven
            accuracy-value 99.12
        }
    }

    DiscreteKeyProperty dkp1 {
        name "test dataset size"
        description "Number of images in the test dataset"
        priority high
        version 1.0
        status toBeDiscussed
        QuantitativeDatasetAnalysis dstest_data for dataset dstest{
            size 10000
        }
    }

    DiscreteKeyProperty dkp2 {
        name "Digit 7 incorrect recognition"
        description "Number of images of digit 7 have beeing incorrectly
            recognised."
            priority high
            version 1.0
            status toBeDiscussed
            incorrectly recognized QuantitativeDataAnalysis inc_reco for dataset
                dstest {
                equivalenceclasses seven
                amount 9
            }
        }
    }
}
```

Listing 4.14 Quantitative KeyProperties Specification

In Listing 4.15, we show a specification of the qualitative key-properties in the context of our running example. We illustrate the specification of the logical, ordinal and nominal key-properties. In summary, we present some specified key-properties:

- image recognition anomalies of the handwritten digit "7".
- estimated evaluation of the recognition of the handwritten digit "7".
- estimated correctness of the recognition and classification of the equivalence class "7".

```
    KeyProperties{
        QualitativeKeyProperties {
            NominalKeyProperty nkp1 {
                RecognitionAnomaly recoanomly_seven {
                    description "Seven recognised often as Two or One"
                    equivalenceclasses seven
                }
            }

            OrdinalKeyProperty okp1 {
                RecognitionEvaluation recoeval_seven {
                    equivalenceclasses seven
                    recognition-value bad
                }
            }

            LogicalKeyProperty lkp1 {
                EquivalenceClassRecognitionCorrectness erc {
                    recognition-correctness incorrectly
                    classification-correctness correctly
                    equivalenceclasses seven
                }
            }
        }
    }
```

Listing 4.15 Qualitative KeyProperties Specification

4.4 Tool-support

This section presents our tool[7] that supports the specification of the requirements and key-properties with the SEMKIS-DSL. Our tool consists mainly of a textual editor permitting to write a specification with the SEMKIS-DSL. First, we present the features of our textual editor. Then, we present some design choices, implementation and possible tests of the grammar of the SEMKIS-DSL. The source code of our toolkit is available publicly [152].

The SEMKIS-DSL has been developed using the Xtext framework [123] within the Eclipse IDE [153]. Eclipse is an open-source software engineering environment extensible with plug-ins. We developed our grammar in Eclipse by importing the Xtext framework through a plugin. The SEMKIS-DSL grammar has been developed in a .xtext file inside a Xtext project. The grammar consists of numerous rules representing the SEMKIS-DSL syntax. Among others, we defined these types of grammar rules in our grammar; fixed keywords, optional words, relation types, user-selectable keywords . . . It represents the metamodel of all specification models of the neural network's requirements and key-properties. Xtext permits the generation **EclipseEcore** of an EMF Ecore model (.ecore file) [106] and a generator model (.genmodel file) once the grammar definition has been finished. The Ecore model contains most relevant concepts defined in the SEMKIS grammar such as classes, attributes and relation. The genmodel contains the remaining information (e.g. file/path information) required for the generation of some source code required for example to implement model checkers or a model transformation algorithm. This allows us to implement some validation rules (e.g. unique identifiers, number range validations, . . .) These rules shall support the engineers during their specification and to reduce the likelihood of potential specification errors.

The SEMKIS specification models are written within a textual editor. The main feature of our textual editor is the specification of the neural network's requirements and key-properties. The textual editor offers many more useful features to support engineers during their specifications. Among others, the engineers may benefit from features for preventing specifications errors, fastening the specification process, structuring their specifications, opening different views for a better readability of the specification. These features improve the readability, accessibility and validation of the SEMKIS specifications. We grouped the features in these categories:

1. Specification error prevention

[7]The SEMKIS toolkit can be found on Github via the following link: https://github.com/Benji91/lu.uni.lassy.phdthesis.semkis.toolkit.experimentations.

- syntactical validation rules
- syntax highlighting
- quick fixes
- scoping

2. Fast specifications

- templates
- syntax highlighting
- auto-completion

3. Specification structuring

- auto-formatting

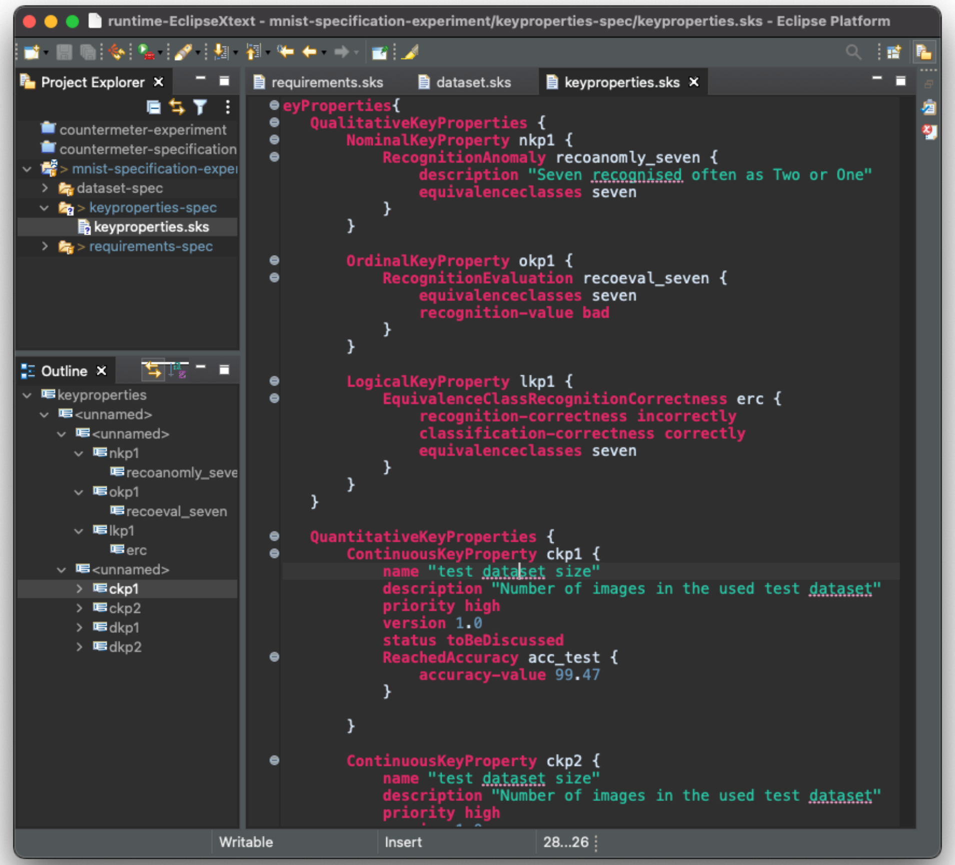
4. Views

- specification outline view
- comparison view (multiple textual editor in one window)

The first feature category, **Specification error prevention**, eases the process of learning the SEMKIS-DSL syntax, permits the validation the specifications and prevents writing errors in the specification. Thus, the engineers shall be able to specify correctly their requirements and key-properties models. The second feature category, **Fast specifications**, eases as well the process of learning the syntax and reduces the time of specifying a model. Templates and auto-completion can help to specify effectively some requirements and key-properties models. The third feature category, **Specification structuring**, supports the engineers to write structured specifications. The resulting specification is readable and information can be easily accessed. Finally, the last features category, **Views**, supports the engineers to access some specifica parts of the specification and review their specifications. These features allow to improve the maintainability of a specification.

Figure 4.4 shows our textual editor with some sample specifications.

Our tool permits also to test a SEMKIS DSL in two ways. The first way is to use unit testing, a feature proposed by Xtext itself. This feature is a unit testing framework typically used in Java [154] application. It is based on JUnit[155]. The software engineer may develop some unit tests to automatically verify some implemented source code (validation rules, model transformations, templates,...) running in the background, while specifying the requirements and key-properties with the SEMKIS-DSL. Moreover, these tests can be



```
eyProperties{
    QualitativeKeyProperties {
        NominalKeyProperty nkp1 {
            RecognitionAnomaly recoanomly_seven {
                description "Seven recognised often as Two or One"
                equivalenceclasses seven
            }
        }

        OrdinalKeyProperty okp1 {
            RecognitionEvaluation recoeval_seven {
                equivalenceclasses seven
                recognition-value bad
            }
        }

        LogicalKeyProperty lkp1 {
            EquivalenceClassRecognitionCorrectness erc {
                recognition-correctness incorrectly
                classification-correctness correctly
                equivalenceclasses seven
            }
        }
    }

    QuantitativeKeyProperties {
        ContinuousKeyProperty ckp1 {
            name "test dataset size"
            description "Number of images in the used test dataset"
            priority high
            version 1.0
            status toBeDiscussed
            ReachedAccuracy acc_test {
                accuracy-value 99.47
            }

        }

        ContinuousKeyProperty ckp2 {
            name "test dataset size"
            description "Number of images in the used test dataset"
            priority high
```

Fig. 4.4 Textual Editor with some sample specifications.

implemented on the textual editor or on the SEMKIS grammar itself. These tests contribute to an improvement of our SEMKIS specification tool by improving maintainability, reliability and availability. Another way of testing the SEMKIS DSL is to specify several requirements and specification for concrete scenarios. These tests contribute to a better visibility the usability of the grammar. The software engineer eases the validation process of the SEMKIS grammar. We have tested the grammar on two concrete examples:

1. MNIST: the recognition of handwritten digits
2. Counter meter experiment: the recognition of a counter meter consisting of two 7-segment digits.

We designed our running examples in this thesis with the MNIST study. The counter meter experimentation is presented in chapter 4 and part of our main experimentation of the SEMKIS methodology.

4.5 Publications

We have written two papers in the context of the SEMKIS domain-specific language. In these papers, we present the SEMKIS-DSL, its conceptual model and concrete syntax:

- The technical report[8] titled 'SEMKIS-DSL: a Domain-Specific Language for Specifying Neural Networks' Key-Properties' [2] presents the initial version of the SEMKIS-DSL containing only the initial conceptual model and concrete syntax.
- The technical report[9] titled 'SEMKIS-DSL: A Domain-Specific Language for Improving Requirements Engineering of Neural Networks' [156] presents the updated SEMKIS-DSL as presented in this thesis. It introduces the complete SEMKIS-DSL (conceptual model and concrete syntaxe) in the context of a model-driven engineering approach for automatising data generation. The concepts are illustrated using a running example.

This chapter presents briefly the conceptual model and the concrete syntaxe of the SEMKIS-DSL as presented in these papers.

[8]The technical report is part of the Laboratory of Advanced Software Systems (LASSY) at the University of Luxembourg.

[9]The technical report is part of the LASSY, submitted for review at a journal.

Chapter 5

Case study: Counter Meter Recognition

Abstract

This chapter presents a complete case study on the SEMKIS methodology in order to experiment the SEMKIS process and SEMKIS-DSL. We conducted the experimentation on the meter counter recognition problem[78]. Firstly, we introduce our version of the meter counter recognition problem. We simulate an invented scenario with some company, who requires a neural network for their internal needs. Secondly, we instantiate and present the first iteration of the SEMKIS process. Each performed activity is presented including the engineering of datasets; the engineering of our neural network; the analysis of the neural network's recognition skills;. . . Additionally, we show some concrete specifications of the neural network's requirements and key-properties using the SEMKIS-DSL. Thirdly, we present the second iteration of the SEMKIS process. During this iteration, we present an augmentation of the datasets in order to improve the neural network. Then, we present the analysis of the improved neural network to verify whether it satisfies the customer's requirements. Lastly, we present our promising results and achievements of our case study. We show a successful usage of the SEMKIS process and the SEMKIS-DSL.

5.1 Introduction

In this chapter, we present our case study on the SEMKIS methodology conducted on a concrete scenario. We introduce an academic case study on the recognition of a meter counter state. In this case study, we engineer a neural network and datasets by following the activities of our process. Additionally, we use our domain-specific language (SEMKIS-DSL) to specify the requirements and key-properties[1] for reengineering the datasets and improving the neural network. We experiment our process on the problem of recognising the state of a meter counter. However, we will not cover the recognition of real-world meter counter in this case study. We present a custom meter counter recognition problem in order to focus more on the presentation of the process workflow, the reengineering of the datasets and the improvement of the neural network. The case study used as a basis here, has been published in the ESSE2020 conference [4]. We extend the published case study in this thesis by including some specifications with our SEMKIS-DSL. The source codes (process execution and DSL specifications) of this case study are publicly available [152, 157].

The aim of this case study is to improve a trained neural network by analysing its recognition skills and retraining it on a reengineered dataset. We want to show that our data engineering method helps to improve neural network in order to satisfy the requirements of a customer[2]. This experimentation shall show :

- Successful improvement of the neural network.
- Validation of the neural network.
- Successful execution of our SEMKIS process.
- That our domain-specific language is useful for understanding and specifying the key-properties.
- Successful execution of our method to define a dataset augmentation.

The context of our case study is to engineer a dataset and a neural network for recognising a meter counter state. A meter counter [78] is an electronic or mechanical instrument for measuring the amount of something. Among others, these instruments are used to measure the used water, used electricity, the distance . . . Reading these meter counters is sometimes very difficult, since many countries use mechanical meter counters. Digital meter counters are usually read automatically via some wireless or wired network. In many countries, electricity seller companies or water distribution companies are still obliged to read annually the meter counter. We will present the context with an example of a mechanical meter counter for water distribution. Figure 5.1 shows an example of a real world meter counter working mechanically. The state can be read from this meter counter. The state of the meter counter is read to determine the consumption of the water. Basically, every year an employee of the

[1]Reminder: Key-properties describe the learnt recognition skills of a neural network after its training.

[2]Note that no real customer was involved in this experimentation. The customer is only an actor representing a person, who wants to order a neural network.

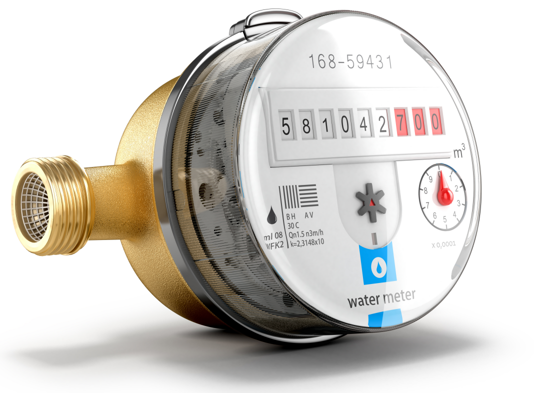


Fig. 5.1 Sample photo of a meter counter (©[sveta] / Adobe Stock).

water distribution company (or a municipal worker) is going to each house in order to read the state of the meter counter. The current state is then subtracted with the state noted last year. The resulting water consumption is often multiplied once with the price of the water per cubic meter and with the price of wastewater.

It is clear that the price highly depends on the employee's recognised value and the previously stored value. The process itself is very time-consuming and prone to human error. There are several factors, that could lead to a reading error of the meter counter:

- Light conditions. (e.g. darkness)
- Age of the meter counter. (e.g. faded font)
- Visual perception of the employee reading the value. (e.g. vision strength)
- Mental state of the employee reading the value. (e.g. tired, family issues, ill,...)
- Mistakes while writing down the state.
- Unreadable handwritten state by the accountants.
- Manual data transfer from customer to company via paper. (e.g. lack of digitalisation)

These mistakes may lead to financial problems to the companies. Mistakes in general costs much money, due to the time investing to correct manually these mistakes. We list some human failures that could cause problems within the company:

- High price of the wasted water due to reading errors, leading to angry customers.
- Low price of the wasted water due to reading errors, leading to financial issues to the company.
- Time loss while reading, which lead to high expenses.

Due to these factors and potential mistakes, a software can help to read these meter counters and automatise many manual tasks. We think that a neural network shall support the employees to read these meter counters. The neural network shall be able to recognize the state of meter counters in various lightning conditions, clean/dirty meter counter or faded digits. A possible software could be a phone application that is able to read the meter counter using a neural network and connected to the companies servers. Thus, the app could automatically transmit the photo and the recognized state to the servers for the accountants. The neural network shall reduce the costs made from mistakes and read the state of the meter counter more reliably than humans.

Our case study presents an instance of our iterative process for augmenting a dataset to build a neural network, which recognise the state of a two-digit mechanical meter counter[3]. Note that, we do not cover here the recognition of real-world meter counter states. We cover only the recognition of two-digit mechanical meter counters. These meter counters consists of two 7-segment digits representing the state of a meter counter. We simplify this problem to provide a better understanding of the workflow of SEMKIS process and the domain-specific language. The case study can be summarised in these two process iteration:

- First process iteration.
    1. Specify precisely the requirements.
    2. Engineer the datasets.
    3. Engineer and train the neural network.
    4. Specify and analyse its key-properties.
    5. Define and perform a dataset augmentation.
- Second process iteration.
    1. Reengineer the requirements (if necessary).
    2. Reengineer the datasets.

[3]We simplified the problem of recognising a real-world meter counter for illustrative purposes of our methodology.

3. Reengineer or retrain the neural network.
4. Specify and analyse its key-properties.
5. Validate the neural network or redefine/perform a dataset augmentation.

Even if our problem has been simplified, the images of the meter counters shall mimic the problems of real-world meter counters. Therefore, we defined to have different categories of images to represent meter counters :

- Clear/Sharp images.
- Dirty images.
- Bright/Dark images.
- Shifted digits due to the mechanical movement.
- Images with combinations of the above properties (1.-4.).

Figure 5.2 shows some sample images of meter counter states.

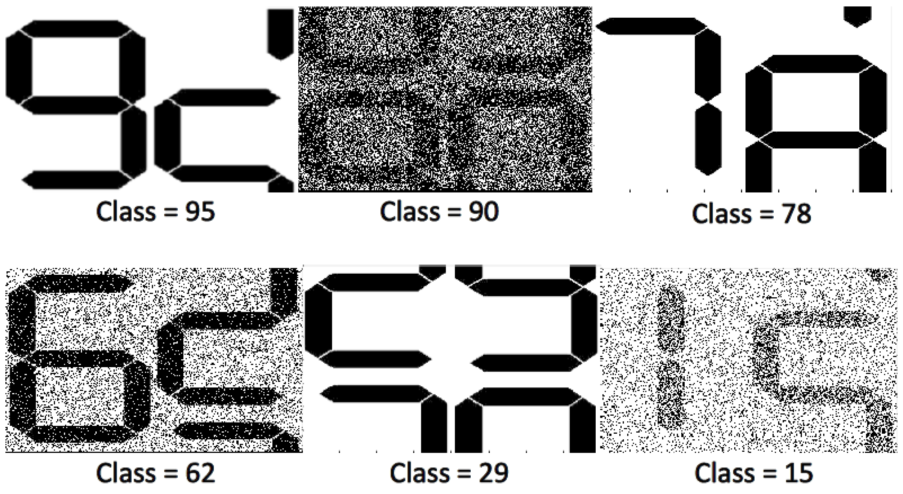


Fig. 5.2 Samples of some metercounter images.

In the next sections, we illustrate the different activities of the SEMKIS process in the context of the meter counter recognition problem. We show the engineering of custom images and improve a neural network by reengineering datasets. We show some augmented datasets, reengineering with the help of specification with the SEMKIS-DSL. We present some custom images of meter counters, engineered datasets, a neural network, requirements and key-property specification, Finally, we present the results of our experimentation in the conclusion.

5.2 1st process iteration: engineering a neural network

In this section, we instantiate a 1st iteration of the SEMKIS process as presented in section 3.6 on the counter meter case study. We present our case study that we conducted on our SEMKIS methodology. Since our methodology has been designed as an iterative process consisting of different activities, we will present two iterations of this process. The case study focuses on the meter counter recognition problem. We engineer some datasets required for building a neural network that is capable of recognising the state of a two-digit meter counter. We want to show a successful usage of the SEMKIS process by improving a neural network. These improvements shall be achieved by performing the process' activities and using the SEMKIS domain-specific language. First, we start with the requirements engineering phase in the next section.

5.2.1 Pre-activity - Requirements engineering

In this section, we instantiate the SEMKIS preliminary-activity, presented in section 3.6.1, by defining the project settings (e.g. functional and non-functional requirements) in the context of the meter counter case study. During the requirement engineering phase, we simulate a discussion in between a software engineer and a customer[4]. During the meeting, the software engineer discusses the needs with the customers. We defined a list of needs that we will use as a base to specify the customer's requirements. Among others, we picked the most relevant customer's needs to illustrate this task.

The customer expresses the following needs concerning the neural network:

1. Shall recognise the state of a two-digit meter counter.
2. Shall recognise the state in different lightning conditions.
3. Shall recognise the state if the meter counter is a bit dirty or dusty.
4. Shall recognise the state in different lightning conditions.
5. Shall display the state on a screen.
6. Our meter counter displays two digital digits representing the state of a meter counter.
7. The meter counter are mechanical devices and are rotating the digits while counting the consumption.

[4]Note that no real customer is involved during this case study. We only use this actor for illustrating the process runtime.

Before moving to the activities of our process, we specify the initial customer's requirements. Our goal is to describe these requirements as precise as possible to correspond to the targeted end-product (our target neural network). We analysed the discussion with the customer and the defined needs to deduce the requirements.

Our SEMKIS-DSL supports the specification of the requirements related to the datasets required for training and testing the neural network. We will use the specification of the dataset requirements to engineer our raw dataset. Listing 5.1 shows a partial specification of some equivalence classes. Note that the full specification is presented in the appendix under section B.1.1.

Listing 5.2 shows a partial specification of some data elements. We show the specifications concerning the unique digits '0' and '1'. Note that our meter counter states consists of two digits, which can be constructed out of all unique digits. These data elements describe each possible unique digit, that can be used either on the ones or on the tens position. The full specification is presented in the appendix under section B.1.2.

Listing 5.3 shows a partial specification of the datasets. We specified the different dataset required for training and testing the target neural network. In summary, we describe the format, the size, the dataset type and the contained equivalence classes. A larger specification is presented in the appendix under section B.1.5.

Listing 5.4 shows a specification of the nonfunctional requirements. The specified nonfunctional requirements describe the recognition values (e.g. accuracy, loss...) of the target neural network. We defined the minimal and maximal targeted recognition values that are acceptable for our target neural network. Moreover, we specified an example for the targeted recognition values of an equivalence class. A larger specification can be found in the appendix under section B.4.

Listing 5.5 shows a partial specification of the functional requirements for our case study. We specified the target neural network's expect input and output. Our aim is to show what shall be recognised by the target neural network concretely. We specified the data elements required for each input including details such as the concrete input values. We also specified the output, such as the size, the output values and the corresponding equivalence class for each output neuron. A larger specification can be found in the appendix under section B.5.

Thanks to the SEMKIS approach, we defined an activity dedicated for the engineering of the requirements. We executed successfully the activity and defined many requirements with the customer. Moreover, we showed a successfull specification of the functional and non-functional requirements using the SEMKIS-DSL. While most requirements can be specified with our domain-specific language, it is still possible that the DSL shall be extended. Therefore, we designed our grammar to be maintainable and extensible to add additional features.

```
        EquivalenceClasses {
            EquivalenceClass LeftDigit {
                name "equivalence classes of the Left Digit"
                EquivalenceClass LeftZero {
                    name 'Left-Digit-Zero-Images'
                    value '0'
                    acceptance-rule greaterThan 0.95
                    data-elements imgDigitZero
                }
                EquivalenceClass LeftOne {
                    name 'Left-Digit-One-Images'
                    value '1'
                    acceptance-rule greaterThan 0.95
                    data-elements imgDigitOne
                }
                EquivalenceClass LeftTwo {
                    name 'Left-Digit-Two-Images'
                    value '2'
                    acceptance-rule greaterThan 0.95
                    data-elements imgDigitTwo
                }}
            EquivalenceClass RightDigit{
                name "equivalence classes of the Right Digit"
                EquivalenceClass RightZero {
                    name 'Right-Digit-Zerp-Images'
                    value '0'
                    acceptance-rule greaterThan 0.95
                    data-elements imgDigitZero
                }
                EquivalenceClass RightOne {
                    name 'Right-Digit-One-Images'
                    value '1'
                    acceptance-rule greaterThan 0.95
                    data-elements imgDigitOne
                }
                EquivalenceClass RightTwo {
                    name 'Right-Digit-Two-Images'
                    value '2'
                    acceptance-rule greaterThan 0.95
                    data-elements imgDigitTwo
                }}}
```

Listing 5.1 Equivalence Class Specification

```
DataElements {
    DataElement imgDigitZero {
        name 'Digit Zero'
        description 'Image of a 7-segment black "0" on white
            background'
        image-elements {
            ImageElement img0{
                format [310 x 165]
                pixel-format rgb
                pixel-value-intervals [0;255]
                image-content {
                    ImageContent img0digit {
                        name 'Digit Zero'
                        description 'black, 7-segment and digital
                            digit "0"'
                        position center
                        pixel-value-intervals [0;10]
                        features shift([up , 10])
                        custom-features 'black shape'
                    }
                    ImageContent img0background {
                        name 'Background'
                        description 'white background'
                        position background
                        pixel-value-intervals [240;255]
                        custom-features 'white background'
                    }}}}
    }

    DataElement imgDigitOne {
        name 'Digit One'
        description 'Image of a 7-segment black "1" on white
            background'
        image-elements {
            ImageElement img1{
                format [310 x 165]
                pixel-format rgb
                pixel-value-intervals [0;255]
                image-content {
                    ImageContent img1digit {
                        name 'Digit One'
                        description 'black, 7-segment and digital
                            digit "1"'
                        position center
                        pixel-value-intervals [0;10]
                        features shift([up , 10])
                        custom-features 'black shape'
                    }
                    ImageContent img1background {
                        name 'Background'
                        description 'white background'
                        position background
                        pixel-value-intervals [240;255]
                        custom-features 'white background'
                    }}}}}
}
```

Listing 5.2 DataElements Specification

```
Datasets {
    ...
    Dataset dstest for process-iteration 0 {
        name 'testing dataset'
        description 'The trained target neural network shall be tested on
             the testing dataset.'
        dataset-type testing
        version 0.1
        size 172
        format [ 310 x 330 x 172]
        equivalenceclasses{
            LeftDigit.LeftZero      (numElements 18),
            LeftDigit.LeftOne       (numElements 10),
            LeftDigit.LeftTwo       (numElements 14),
            LeftDigit.LeftThree     (numElements 18),
            LeftDigit.LeftFour      (numElements 22),
            LeftDigit.LeftFive      (numElements 15),
            LeftDigit.LeftSix       (numElements 19),
            LeftDigit.LeftSeven     (numElements 16),
            LeftDigit.LeftEight     (numElements 17),
            LeftDigit.LeftNine      (numElements 23),
            RightDigit.RightZero    (numElements 20),
            RightDigit.RightOne     (numElements 11),
            RightDigit.RightTwo     (numElements 21),
            RightDigit.RightThree   (numElements 17),
            RightDigit.RightFour    (numElements 8),
            RightDigit.RightFive    (numElements 20),
            RightDigit.RightSix     (numElements 23),
            RightDigit.RightSeven   (numElements 12),
            RightDigit.RightEight   (numElements 16),
            RightDigit.RightNine    (numElements 24)
        }
    }
}
```

Listing 5.3 Dataset Specification

```
nonfunctional Requirements {
    ....
    Requirement NFR3 {
        description 'Global accuracy on testing dstest'
        purpose 'Defining the tolerated accuracy on the testing dataset'
        priority high
        TargetAccuracy tarGlobalTestAcc for dataset dstest{
            min-accuracy 95
            max-accuracy 100
            target-accuracy 99.7
        }
    }

    Requirement NFR6 {
        description 'Global loss on testing dstest'
        purpose 'Defining the tolerated loss on the testing dataset'
        priority high
        TargetLoss tarGlobalTestLoss for dataset dstest{
            min-loss 0.009
            max-loss 0.02
            target-loss 0.15
        }
    }


    Requirement NFR8 {
        description 'Accuracy on testing dataset for the equivalence
            class LeftOne'
        purpose 'Defining the tolerated accuracy for the equivalence
            class LeftOne from the testing dataset'
        priority high
        TargetAccuracy tarEcLeftOneAcc for dataset dstest{
            equivalenceclasses LeftDigit.LeftOne
            min-accuracy 95
            max-accuracy 100
            target-accuracy 97
        }
    }

    Requirement NFR8 {
        description 'Accuracy on testing dataset for the equivalence
            class LeftOne'
        purpose 'Defining the tolerated accuracy for the equivalence
            class LeftOne from the training dataset'
        priority high
        TargetAccuracy tarEcLeftOneAcc for dataset dstrain{
            equivalenceclasses LeftDigit.LeftOne
            min-accuracy 99
            max-accuracy 100
            target-accuracy 99.9
        }
    }
...
}
```

Listing 5.4 NonFunctional requirements

```
functional Requirements {
        NeuralNetworksInput {
            neurons-size 102300
            neurons-format [310,330]
            input-data {
                InputData inputZeroZero{
                    neuron-values [0;255]
                    data-elements imgDigitZero,imgDigitZero
                }
                InputData inputZeroOne{
                    neuron-values [0;255]
                    data-elements imgDigitZero,imgDigitOne
                }
                InputData inputZeroTwo{
                    neuron-values [0;255]
                    data-elements imgDigitZero,imgDigitTwo
                }
                ....
            }
        }

        NeuralNetworksOutput {
            neurons-size 20
            neurons-format [1,20]
            output-value-range [0;1]
            output-neurons [
                LeftDigit.LeftZero, LeftDigit.LeftOne, LeftDigit.LeftTwo,
                    LeftDigit.LeftThree, LeftDigit.LeftFour,
                LeftDigit.LeftFive, LeftDigit.LeftSix, LeftDigit.LeftSeven,
                    LeftDigit.LeftEight, LeftDigit.LeftNine,
                RightDigit.RightZero, LeftDigit.LeftOne, RightDigit.RightTwo,
                     RightDigit.RightThree, RightDigit.RightFour,
                RightDigit.RightFive, RightDigit.RightSix, RightDigit.
                    RightSeven, RightDigit.RightEight, RightDigit.RightNine
            ]
            input-data inputZeroZero,inputZeroTwo, inputZeroThree, input
                Zerofour,inputZeroFive,inputZeroSix,inputZeroSeven
        }
    }
```

Listing 5.5 Functional requirements

5.2.2 Activity A - Engineering Raw Datasets

In this section, we instantiate activity A (engineering raw datasets) of the SEMKIS process, presented in section 3.6.2. We design and build the raw datasets representing images of meter counter states.

From the specification of the requirements, we deduce how to design and build our dataset. In order to build the dataset, we require a set of classified reference images. These reference images will serve as Data Input of the SEMKIS process. In addition to the reference images, we simulate that a customer provided a set of images used for testing the targetNN.

Thus, the process' Data Input consists of :

- 10 reference images, $imgs_{ref} = [0,255]^{310\times 165}$, representing the digital digits in $0,..,9$ and classified into 10 equivalence classes, $ec_{ref} = \{0,...,9\}$.
- 172 testing images, $imgs_{test} = [0,255]^{310\times 330}$, classified into 100 equivalence classes $ec_{counterStates} = \{ec_{ref}, ec_{ref}\}$

Thus, the equivalence classes are defined as $ec_{counterStates} = \{ec_{ref}, ec_{ref}\}$. Figure 5.3 shows the 10 reference images.

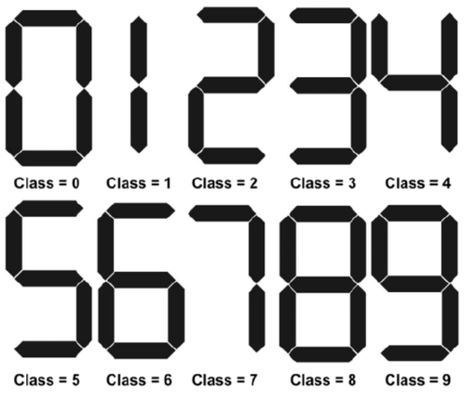


Fig. 5.3 Reference images.

Each reference images represent a unique digit from the set $0,..,9$. We concatenate all possible combinations of two reference images to engineer the images for the training and development datasets. The images shall contain two digits representing a digital number in $[00..99]$. We modify the concatenated images by satisfying the following properties :

- Sharp images without additional modifications.
- Shifted digits to represent the mechanical movement of a meter counter.

- Noisy images representing dusty images.
- Brightness adjustment of the whole image representing lightning issues.
- Brightness adjustment of the digits itself to represent fading of the digits.

Figure 5.4 shows some samples of generated meter counter images.

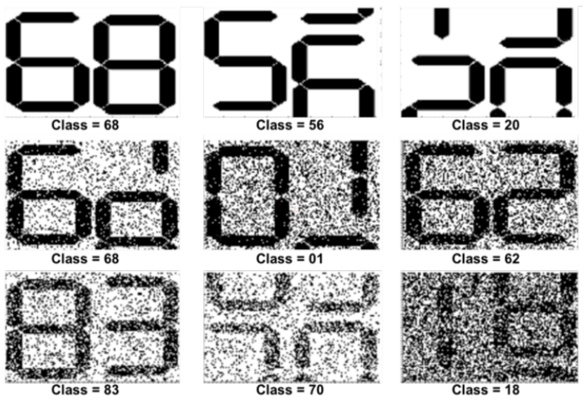


Fig. 5.4 Meter counter sample images.

The finally generated images are classified and put in the raw dataset ds_{raw}. The next step is to construct the training, development and testing dataset. Concerning the testing dataset, we reuse the images provided by the customer. Concerning the training and the development dataset, we selected randomly a bunch of images of the raw dataset. The selected images have been placed into the training and development dataset. The resulting datasets are :

- The training dataset, $ds_{train} = \mathscr{P}([0,255]^{310\times165} \times ec_{counterStates})$, consists of 645 random images from ds_{raw}.
- The development dataset, $ds_{dev} = \mathscr{P}([0,255]^{310\times165} \times ec_{counterStates})$, contains the 40 remaining images from ds_{raw}.
- The testing dataset, $ds_{test} = \mathscr{P}([0,255]^{310\times165} \times ec_{counterStates})$, contains the classified test images.

Unlike traditional ad-hoc approaches, SEMKIS requires the definition of precise data selection criteria deduced from the customer's requirements to ease the selection of optimal data for training and testing neural networks. These data selection criteria can help to improve dataset engineering activities in time and costs, because we avoid inappropariate datasets leading to inaccurate neural network as well as minimize the number of dataset reengineering iterations. Thanks to the SEMKIS process and our requirements' specification, we have successfully designed and built our three raw datasets (training, testing and development dataset). Thus, we were able to efficiently select the appropriate data for our datasets and design datasets that satisfy the customer's requirements.

5.2.3 Activity B - Engineering the targetNN

In this section, we instantiate activity B (engineering a targetNN) of the SEMKIS process, presented in section 3.6.3. We design and implement a neural network architecture that is able to learn to recognise the state of a meter counter.

In this process' activity, we focus on the development of our target neural network. We analysed the requirements to deduce the most appropriate target neural network, that shall be able to recognise the two-digit meter counter. Therefore, we decided to design our targetNN as a convolutional neural network inspired from Laroca-et-al [78]. The CNN architecture has been implemented in Python[145] using Keras framework[44]. The training and testing process of the neural network is executed with the help of the Tensorflow libraries[46].

The architecture of our targetNN is composed of the following components:

- 1 **Input layer** permitting the neural network to read a 310×330 counter meter image.
- 1 **2-Dimensional convolutional layers** applying 64 filters with a 5×5 on the counter meter images from the input to create additional image variations.
- 1 **2-Dimensional convolutional layers** applying again 64 filters with a 5×5 on the filtered counter meter images from the previous layer.
- 1 **2-Dimensional max pooling layer** with a 2×2 kernel for building a feature map.
- 1 **Activation Layer** with ReLu activation function.
- 1 **2-Dimensional convolutional layers** applying again 64 filters with a 3×3 on the features maps to create different variations.
- 1 **2-Dimensional convolutional layers** applying again 64 filters with a 3×3.
- 1 **2-Dimensional max pooling layer** with a 2×2 kernel for rebuilding a feature map.
- 1 **Activation Layer** with ReLu activation function.
- 1 **Flatten Layer** to transform the 2D image into a 1D vector in order to be readable by the Hidden Layer.
- 1 **Hidden Layer** with 128 neurons and the ReLu activation function.

- 1 **Dropout Layer** with 30% dropout of the Hidden layer's neurons to reduce over- and underfitting[148].
- 1 **Hidden Layer** with 64 neurons and the ReLu activation function.
- 1 **Dropout Layer** with 30% dropout of the Hidden layer's neurons to reduce over- and underfitting[148].
- 1 **Output Layer** with 20 neurons and the Sigmoid activation function.

We estimate that our targetNN shall be able to recognise the input images and output the corresponding meter counter state. The input layer takes as input the images of the meter counters. The output layer outputs a probability distribution over the 20 equivalence classes. The probability distribution describes the likelihood that the left (resp. right) digit belongs to one of the first 10 (resp. last 10) equivalence classes. The maximal probability associated to one of the first 10 and last 10 equivalence classes will be chosen as the recognised equivalence class. Finally, the weights on the hidden layers are randomly initialised, and they are updated during the training of the targetNN. We decided to select the function to compute the recognition error. This function is called binary cross-entropy [158], as our loss function. We have selected this loss function as it is used in machine learning for classification problems and it is well suitable for image classification problems [159]. Listing 5.6 shows a snapshot of the PyCharm Development IDE[160] with the implementation of our targetNN in Python.

Unlike in traditional ad-hoc approaches, SEMKIS proposes to deduce the selection of a neural network architecture and its parameters from a requirements specification. The requirements' specification shall help to reduce time and costs while engineering the neural network architecture. Thanks to the SEMKIS process and our requirements' specification, we have successfully engineered a neural network architecture that can be trained on our raw datasets. Hence, the requirements' specification provided us with information (e.g. the targetNNinput, targetNNoutput and the data elements) to efficiently select the appropriate architecture. However, we do not address the problem of designing an optimal architecture, chosing the correct hyperparameters, selecting the optimal layers and selecting the optimal activation function. These problems still remain open issues as we consider a neural network as a black box in this thesis.

```
    def build_cnn(self, input_shape=(310, 330, 1), nb_filter=64):
        """
        Build a completely new CNN model.
        :return:
        """
        lyr_input = layers.Input(shape=input_shape)

        lyr_1_conv_1 = layers.Conv2D(nb_filter, (5, 5), padding='same')(lyr_input)
        lyr_1_con_2 = layers.Conv2D(nb_filter, (5, 5), padding='same')(lyr_1_conv_1)
        lyr_1_pool = layers.MaxPooling2D(pool_size=(2, 2))(lyr_1_con_2)
        lyr_1_act = layers.ReLU()(lyr_1_pool)

        lyr_2_conv_1 = layers.Conv2D(2 * nb_filter, (3, 3))(lyr_1_act)
        lyr_2_conv_2 = layers.Conv2D(2 * nb_filter, (3, 3))(lyr_2_conv_1)
        lyr_2_pool = layers.MaxPooling2D(pool_size=(2, 2))(lyr_2_conv_2)
        lyr_2_act = layers.ReLU()(lyr_2_pool)

        lyr_3_flat = layers.Flatten()(lyr_2_act)

        lyr_4_dense = layers.Dense(2 * nb_filter, activation="relu")(lyr_3_flat)
        lyr_4_dropout = layers.Dropout(rate=0.3)(lyr_4_dense)

        lyr_5_dense = layers.Dense(nb_filter, activation="relu")(lyr_4_dropout)
        lyr_5_dropout = layers.Dropout(rate=0.3)(lyr_5_dense)

        lyr_6_dense = layers.Dense(self.number_of_outputs, activation='sigmoid')(
            lyr_5_dropout)

        model = models.Model(inputs=lyr_input, outputs=lyr_6_dense)
        print(model.summary())
        return model
```

Listing 5.6 TargetNN Implementation as convolutional neural network in Python.

5.2.4 Activity C - Train the targetNN

In this section, we instantiate activity C (training a targetNN) of the SEMKIS process, presented in section 3.6.4. During this activity, we train our implemented neural network architecture to recognise the state of a meter counter. The machine used for training the targetNN has the following specs: Intel i9-7900X (10x3.3GHz); 128GB DDR4-RAM; 2 Nvidia RTX2080TI graphic cards. The targetNN has been trained during 50 epochs and lasted about 4 hours on the 2 graphic cards. During the training, the dataset has been shuffled after each epoch to avoid over- or underfitting. During each epoch, we computed the accuracy and loss after a batch of 25 images has been processed. This allowed us to verify the targetNN's recognition during the training and adjust the architecture's parameters. Figure 5.5 shows the evolution of the targetNN's accuracy and loss on the training and development dataset.

In summary, our targetNN reached the following accuracies and losses on the training and development dataset:

- ds_{train}: accuracy 100% and loss 0.0027.
- ds_{dev}: accuracy 99.37% and loss 0.0240.

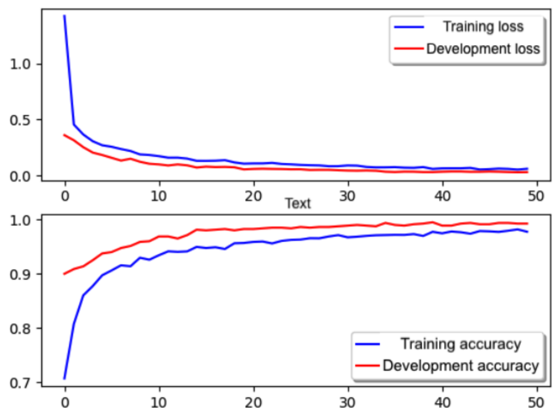


Fig. 5.5 Training diagram of our targetNN.

Listing 5.7 shows an implementation for starting the targetNN's training. The two functions are part of a class, called 'CNN', implemented in Python. The 'initializer' function initialises the attributes of the class, which are used in the training function. The function, called 'train', executes the training of the targetNN.

Thanks to the SEMKIS process, we were able to successfully train our targetNN on the previously engineered training dataset. Moreover, we have successfully used our development data to show a successfull training. The neural network was able to successfully process the training and development data and acquire some recognition skills. These results give us first indicators that we have developed appropriate datasets to train a neural network and an NN architecture that is able to learn.

```python
def __init__(self, x_train, x_dev, x_test, y_train, y_dev, y_test):
    """"
    Initialize the CNN class.
    """
    # Hyper-Parameters
    self.BATCH_SIZE = 16
    self.NUM_EPOCH = 25
    self.number_of_outputs = 20
    self.verbose = 1
    self.shuffling = True
    self.cnn = self.build_cnn()

    # Datasets
    self.x_train = x_train
    self.y_train = y_train
    self.x_dev = x_dev
    self.y_dev = y_dev
    self.x_test = x_test
    self.y_test = y_test

    # Compiler Configuration
    self.cnn.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy
        '])

def train(self, filename):
    """
    Train any CNN Model on a number of epochs
    :return:
    """
    # Train the neural network without data augmentation i obtained an accuracy of
        0.98114
    history = self.cnn.fit(
        self.x_train, self.y_train,
        batch_size=self.BATCH_SIZE,
        epochs=self.NUM_EPOCH,
        validation_data=(self.x_dev, self.y_dev),
        verbose=self.verbose,
        shuffle=self.shuffling
        )

    networkutils.save_model(self.cnn,configutils.configSectionMap("FOLDERS")['
        cnn_path'] + filename)
    analysisutils.plot_cnn_training_curve(history)
    return history
```

Listing 5.7 TargetNN training implementation in Python.

5.2.5 Activity D - Analyse the training monitoring data

In this section, we instantiate activity D (analysing the training monitoring data) of the SEMKIS process, presented in section 3.6.5. During this activity, we analyse the training and acquired recognition skills of our targetNN achieved on our training and development dataset.

After the training, we analyse the training monitoring data computed during the NN training process. The training monitoring data consists of the following information:

- Training curve containing the evolution of the accuracy and loss on the training and development dataset.
- Reached accuracy value on the training and development dataset
- Reached loss value on the training and development dataset

These accuracy and loss values are computed by built-in Keras/Tensorflow functions. They serve to evaluate the training of the model. Figure 5.5 shows the evolution of the accuracy and loss on the training and development dataset. We observed that the accuracy on the development and training dataset increases continuously and the loss decreases continuously in time. Since the recognition of the development dataset is not getting worse than the recognition of the training dataset, we can conclude that it is very unlikely that the targetNN is overfitting. In summary, we reached on the training dataset, ds_{train}, an accuracy of 100% and a loss of 0.0027; and the development dataset, ds_{dev}, an accuracy of 99.37% and a loss of 0.0240. Since the development dataset contains data that differ from the training dataset, it is expected that the accuracy and loss are lower on the development dataset than on the training dataset. The targetNN is basically tested on data, which have not been used for the training. While the targetNN achieved similar accuracies in both cases, the loss is 10 times higher for on the development dataset. In other words, the targetNN is able to correctly recognise the images of both datasets, but the targetNN's certitude of its recognition is about 10 times lower on the development dataset. Since we are using the binary cross entropy function (see Keras [44]), a perfect value would be 0.0. This means that the loss difference can be neglected because the targetNN achieved very low loss values in both cases. We can conclude that the targetNN is performing well on both datasets. Therefore, we decide to stop the training and freeze the targetNN's architecture.

Finally, we have been able to successfully train our targetNN on our training dataset. Moreover, our development dataset showed that the training has been successfully and that the neural network is able to recognise the meter counter on both datasets.

5.2.6 Activity E - Test the targetNN

In this section, we instantiate activity E (testing our targetNN) of the SEMKIS process, presented in section 3.6.6. During this activity, we test our targetNN on our raw datasets to acquire data about its recognition skills.

During this activity, our targetNN processes the training, development and testing dataset. The targetNN takes as input the images of the different datasets and outputs a probability distribution over 20 equivalence classes. The left digit of the image is classified in one of the first 10 equivalence classes with the highest probability is selected. The right digit of the image is classified in one of the last 10 equivalence classes with the highest probability is selected. The selected equivalence classes represent the recognized values of the targetNN.

The three datasets contain different amounts of images, presented in this list :

- The training dataset, ds_{train}, consists of 645 images.
- The development dataset, ds_{dev}, consists of 40 images.
- The testing dataset, ds_{test}, consists of 172 images.

Table 5.1 Confusion Matrix for the recognition of the left digit.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	**15**	0	0	0	0	*1*	0	0	*1*	*1*
1	0	**10**	0	0	0	0	0	0	0	0
2	0	0	**14**	0	0	0	0	0	0	0
3	0	0	*1*	**17**	0	0	0	0	0	0
4	*1*	0	0	0	**20**	*1*	0	0	0	0
5	0	0	0	0	0	**15**	0	0	0	0
6	0	0	0	0	0	0	**19**	0	0	0
7	0	0	0	0	0	0	0	**16**	0	0
8	0	0	0	0	*1*	0	0	0	**15**	*1*
9	0	0	0	0	0	0	0	0	0	**23**

Table 5.2 Confusion Matrix for the recognition of the right digit.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	**20**	0	0	0	0	0	0	0	0	0
1	0	**11**	0	0	0	0	0	0	0	0
2	0	0	**20**	0	0	0	*1*	0	0	0
3	*1*	0	0	**16**	0	0	0	0	0	0
4	0	0	0	0	**8**	0	0	0	0	0
5	0	0	0	0	0	**20**	0	0	0	0
6	0	0	0	0	0	0	**23**	0	0	0
7	0	0	0	0	0	0	0	**12**	0	0
8	0	0	0	0	0	0	0	0	**16**	0
9	*1*	0	0	0	0	0	0	0	0	**3**

These images of all datasets have been classified manually. After having executed the tests on the targetNN and having collected the targetNN's recognitions, we compared the recognized equivalence classes with the targetNN's classifications. Thanks to the comparison, we filtered the images into four categories:

- Correctly classified and correctly recognised.
- Correctly classified and incorrectly recognised.
- Incorrectly classified and correctly recognised.
- Incorrectly classified and Incorrectly recognised.

We used the bokeh data visualisation library [53] to collect more information about the targetNN's recognition. This information is called test monitoring data, and consists of:

- accuracy and loss values on the datasets
- confusion matrices containing the amount of correctly and incorrectly recognized images (CR and ICR images).
- 2 grids of correctly and incorrectly recognized images.

We show some confusion matrices of the recognition of the test images for the left and right digit in tables 5.1 and 5.2. Figure 5.6 shows incorrectly recognised images including information about the image's classification, recognition and targetNN's output (probability distribution for each digit).

Table 5.3 shows the accuracies and losses reached on the different datasets.

Table 5.3 Accuracies and Losses on the different datasets

Datasets			
	ds_{train}	ds_{dev}	ds_{test}
accuracy	100%	99.37%	99.56%
loss	0.002707	0.024017	0.028118

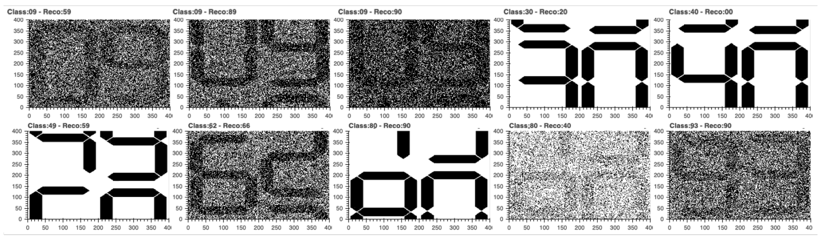


Fig. 5.6 Some incorrectly recognized images

Listing 5.8 shows the implementation of the testing function for the targetNN.

```python
def evaluate_model(self, filename):
    """
    Used for Evaluating a CNN Model
    :param filename:
    """
    f = open(configutils.configSectionMap("FOLDERS")['cnn_performance_path'] +
        filename, "w+")
    phases = ["training", "validation", "testing"]
    for phase in phases:
        if phase == "training":
            score = self.cnn.evaluate(self.x_train, self.y_train, verbose=0)

        if phase == "validation":
            score = self.cnn.evaluate(self.ax_dev, self.y_dev, verbose=0)

        if phase == "testing":
            score = self.cnn.evaluate(self.x_test, self.y_test, verbose=0)

        print(f"Evaluation of Model on {phase.capitalize()} dataset")
        print("%s: %.2f%%" % (self.cnn.metrics_names[1], score[1] * 100))
        print("%s: %f" % (self.cnn.metrics_names[0], score[0]))
        f.write(f"Evaluation of Model on {phase.capitalize()} Dataset\r\n")
        f.write("==============================================\r\n")
        f.write("%s: %.2f%%\r\n" % (self.cnn.metrics_names[1], score[1] * 100))
        f.write("%s: %f\r\n" % (self.cnn.metrics_names[0], score[0]))
        f.write("==============================================\r\n")
    f.close()
```

Listing 5.8 TargetNN Evaluation as convolutional neural network in Python.

Finally, we have been able to successfully test our targetNN on our testing dataset. We have successfully generated some test monitoring data describing the targetNN's recognition skills. However, we think that they might be other types of test monitoring data not covered by this experimentation. Depending on the context, it might be useful to generate additional data to describing the acquired recognition skills.

5.2.7 Activity F - Analyse the test monitoring data

In this section, we instantiate the different activities of subprocess F (analysing the test monitoring data) of the SEMKIS process, presented in section 3.6.7. In this subprocess, we focus on the analysis of the test monitoring, which includes the specification of the targetNN's key-properties. We present each activity of this subprocess to describe the complete process for providing a key-property specification. We start with the presentation of the first activity, which focuses on the identification of the new key-properties.

5.2.7.1 Activity F.1 - Identify New Key-properties

In this section, we instantiate activity F.1 (Identify observations on test monitoring data) of the SEMKIS process, presented in section 3.6.7.1. During this activity, we specify some observations made on the targetNN's test monitoring data from the previous activity.

To start this activity, we require as input the test monitoring data gathered during the tests of the targetNN. In this activity, we extract the most relevant observation of the test monitoring data that help us to specify the targetNN's key-properties. First, we sort the test monitoring data in order to prepare them for the analysis. We group the test monitoring data into our two main categories :

- Quantitative data[5]
  - Accuracy values
  - Loss values
  - Confusion matrices
- Qualitative data
  - Grids of CR and ICR recognised images
  - Accuracy and Loss evolution diagram

The sorted test monitoring data are used to identify the most relevant observation leading to the targetNN's key-properties. We analysed the test monitoring data, and specified a list of observations. These observations serve as a base to specify the key-properties with

[5]Note that the test monitoring data has been obtained by testing the targetNN on the training, development and testing dataset.

the SEMKIS-DSL. They can be seen as some sort of guidance to specify the correct key-properties. We specified the identified observations into two tables [6]. Table 5.4 illustrates our specified list of quantitative observations. Table 5.5 illustrates our specified list of qualitative observations.

Table 5.4 Specified list of quantitative observations.

Category	**Subcategory**	**Observation**
Quantitative	Discrete	Number of CR/IR images
Quantitative	Discrete	Number of CR/IR images per equivalence class
Quantitative	Discrete	Dataset size
Quantitative	Continuous	Ratio of CR/IR images
Quantitative	Continuous	Ratio of CR/IR images per equivalence class
Quantitative	Continuous	Ratio of CR/IR images per equivalence class
Quantitative	Continuous	Neural network's recognition precision

Table 5.5 Specified list of qualitative observations.

Category	**Subcategory**	**Observation**
Qualitative	Logical	Data and classification correctness
Qualitative	Logical	Correctness of the classification and recognition
Qualitative	Nominal	Data consistency
Qualitative	Nominal	Groups of incorrectly recognised images
Qualitative	Ordinal	Recognition weaknesses and strengths
Qualitative	Ordinal	Threshold for belonging to an equivalence class

[6]Note that these tables has been published in ESSE 2020 [4].

Thanks to the SEMKIS process, we have successfully specified some observation made on the test monitoring data from the previous activity. This activity shows us that we have been able to obtain first insights and indicators of the neural network's acquired recognition skills during the training.

5.2.7.2 Activity F.2 - Specify Key-properties

In this section, we instantiate activity F.2 (Specify key-properties) of the SEMKIS process, presented in section 3.6.7.2. During this activity, we specify the targetNN's recognition skills as key-properties with our SEMKIS-DSL based on the previously specified observations. The previously specified observations serve as a base to determine the key-properties and describe the targetNN's recognition skills. From the observations, we can determine that we are required to specify the two types of key-properties, quantitative and qualitative properties.

Firstly, we present the specification of the quantitative key-properties, consisting of discrete and continuous key-properties. Listing 5.9 contains the specification of some quantitative key-properties. A larger specification can be found in the Appendix B.1.6. We already observe that the targetNN has issues in recognising the left digit of the meter counter state compared to the recognition of the right digit.

Secondly, we present the specification of the quantitative key-properties, consisting of nominal, ordinal and logical keyproperties. Listing 5.10 contains the specification of some qualitative key-properties. A larger specification can be found in the Appendix B.1.7. Again, we can observe that the targetNN has issues in recognising the left digit, but more on images containing noise and shifted digits.

Thanks to the SEMKIS process and the SEMKIS-DSL, we show a successfull specification of the neural network's key-properties. We described successfully some neural network's recognition skills as key-properties with the SEMKIS-DSL. The specification helped us to improve our understanding of the neural network's recognition skills. The presented specification is only representative and does not cover all existing key-properties for all szenarios. Since we focused mainly on image recognition problems, it is possible that some key-properties can't be specified in another domain problems. As future work, it is required to verify whether the current version of the DSL covers most common key-properties. However, we have designed our SEMKIS-DSL grammar to be maintainable and extensible with other kinds of key-properties, if other key-properties are required.

```
KeyProperties{
    QuantitativeKeyProperties {
        ContinuousKeyProperty ckp3 {
            name "testing dataset accuracy"
            description "Number of images in the used test dataset"
            priority high
            version 1.0
            status toBeDiscussed
            ReachedAccuracy acc_test {accuracy-value 99.50}
        }
        DiscreteKeyProperty dkp3 {
            name "Incorrect recognition of digit 8"
            description "NN recognises the left digit on 8 images incorrectly."
            priority high
            version 1.0
            status toBeDiscussed
            incorrectly recognized QuantitativeDataAnalysis c_class for dataset
                dstest {amount 8}
        }
        DiscreteKeyProperty dkp4 {
            name "Incorrect recognition of digit 9"
            description "NN recognises the right digit on 3 images incorrectly.."
            priority high
            version 1.0
            status toBeDiscussed
            incorrectly recognized QuantitativeDataAnalysis c_class for dataset
                dstest {amount 2}}
        }...
}
```

Listing 5.9 Specified quantitative key-properties with the SEMKIS-DSL.

```
KeyProperties{
    QualitativeKeyProperties {
        NominalKeyProperty np3 {
            RecognitionCharacteristic recoChar {
                recognition-ration 60
                characteristics 'Left Zero not recognised on noisy images.'
                equivalenceclasses LeftDigit.LeftZero
            }
        }
        ...
        OrdinalKeyProperty okp1 {
            EquivalenceclassSelection ecSelection {
                selection-function maximum
                selection-threshold greaterOrEqualThan 0.5
            }
        }

        LogicalKeyProperty lkp2 {
            EquivalenceClassRecognitionCorrectness ecIncorrectRecoLeftDigit {
                recognition-correctness incorrectly
                classification-correctness correctly
                equivalenceclasses LeftDigit.LeftZero, LeftDigit.LeftTwo,
                    LeftDigit.LeftFour, LeftDigit.LeftFive,LeftDigit.LeftEight,
                    LeftDigit.LeftNine
            }
        }
        ....
    }
}
```

Listing 5.10 Specified qualitative key-properties with the SEMKIS-DSL.

5.2.7.3 Activity F.3 - Validate targetNN's key-properties

In this section, we instantiate activity F.3 (Analyse the key-properties) of the SEMKIS process, presented in section 3.6.7.3. During this activity, we have analysed our key-properties specification to understand and provide more details about our targetNN's recognition skills. We analyse our quantitative and qualitative properties to determine the recognition strengths and weaknesses of our targetNN. The strengths and weaknesses are described in natural language in order to be presentable and understandable by our customer.

We have compared the specification of the key-properties with the specification of the requirements. If some issues have been detected in one of the specifications, we propose some suggestions to improve the specifications. We detect that some key-properties have not been satisfied. As an example, the specification of the key-properties shows that there are some recognition problems with the left digit. In that case, we need to perform some changes in our dataset to improve the recognition of the left digit. Moreover, we observe that the loss value has not been satisfied. The reached loss is higher than the targeted loss. Thus, we suggest generating additional data and adding them to the training dataset to strengthen the targetNN's recognition. Additionally, we detect some issues in the specification of the requirements. As an example, the test dataset size shall be augmented since we do not have enough testing images to test all equivalence classes. We suggest to the customer to provide more testing images if possible. Moreover, the targetNN's recognition policy shall be more precise to define which equivalence class shall be chosen as the recognised one. Thus, we suggest updating the requirements' specification. We have summarised the weaknesses and strength in two tables. Table 5.6 shows some identified weaknesses of our targetNN. Table 5.7 shows some identified strengths of our targetNN.

Thanks to the SEMKIS process and the key-properties specification, we have successfully learned more about the neural network's acquired recognition skills. We have successfully analysed our key-properties specification and better understood the neural network's recognition skills.

Table 5.6 Identified weaknesses and suggested solutions.

Weakness	Solution
Small test dataset.	Ask customer to augment the test dataset with sufficient image variations and at least 5 images per equivalence class.
Recognition problems of the digits 0 and 9 at the right position.	Augment the training dataset with images that contain a 9 or a 0 at the right position.
Recognition precision (loss value too high).	Augment the training dataset.
Neural network's recognition policy.	New policy: An image belongs to an equivalence class if the recognition probability is higher than 0.9.

Table 5.7 Identified strengths and descriptions.

Strength	Description
Overall recognition correctness.	The targetNN recognised correctly recognise more than 99% of the training, development and testing data.
targetNN's reliability.	The targetNN is not showing signs of over- and underfitting.
targetNN's recognition precision.	The targetNN's recognition precision can be improved to satisfy the initial requirement.
Dataset augmentation simplicity.	The dataset design allows us to generate efficiently synthetic data to augment the dataset.

5.2.7.4 Activity F.4 - Specify improved key-properties

In this section, we instantiate activity F.4 (Specify improved key-properties) of the SEMKIS process, presented in section 3.6.7.4. During this activity, we meet our customer to present and discuss the targetNN and its recognition skills. Moreover, we specify the improved key-properties [7] with him.

The first task consists in discussing the key-properties, strengths and weaknesses of the targetNN with the customer. We simulate a discussion with a customer, who needs a neural network. During the discussion with the customer, we describe the specification of the key-properties, strengths and weaknesses. We present and explain the recognition issues and solution suggestions to the customer. Due to the recognition issues of the left digit, the insufficient number of tests and the recognition policy, the customer did not validate the targetNN. Thus, it is required to perform some changes on the dataset.

Based on the discussion and our suggestions, we suppose that the customer decided not to validate the targetNN. Thus, we are required to reengineer the datasets to improve the targetNN's recognition. Therefore, we specify a set of improved key-properties that shall be achieved after the next process iteration. These key-properties will serve in the next activity to support the data engineer to generate additional synthetic data for reengineering the datasets. Table 5.8 presents some improved key-properties deduced from the discussion with the customer. Ideally, these properties shall be satisfied after the next process iteration.

Thanks to the SEMKIS process and this activity, we have successfully identified some improved key-properties with the customer. We have successfully identified some inacceptable recognition skills that shall be improved during the next process iteration. Moreover, the activity helped to revise some requirements and clarify them according to the customer's needs.

[7]Note that the SEMKIS-DSL does not support the specification of the improved key-properties. Nevertheless, a newer version of the SEMKIS-DSL could be extended to support this specification.

Table 5.8 Improved Keyproperties.

Key-property-ID	Description
KP1	The acceptance threshold that an image belongs to an equivalence class should be fixed to 0.9.
KP2	The testing dataset shall contain at least 300 images. There should be at least 5 images per equivalence class.
KP3	The tolerated number of incorrectly recognised right digits shall be a maximum of 3.

5.2.8 Activity G - Specifying a dataset augmentation

In this section, we instantiate activity G (Specify Dataset Augmentation) of the SEMKIS process, presented in section 3.6.8. During this activity, we specify a dataset augmentation obtained from the customer's feedback of the previous activity.

We start this activity by analysing the discussed improved key-properties to deduce the required dataset augmentation operations. From the discussion with the customer, we can deduce that different changes on the training and testing dataset shall be required. Concerning the training dataset, we shall generate some additional data to augment the dataset. Since most of the left digits have not been recognised correctly, we will generate random counter meter states images. In order to have sufficient variations of these image, we add some brightness and noise variations to the images. The resulting synthetic images shall provide sufficient variation in our dataset. If our approach fails to improve the targetNN's recognition, we focus on generating more images with a 0 or a 9 to strengthen the targetNN's recognition. The testing dataset shall be extended with additional data, which shall be provided by the customer, if possible.

Table 5.9 illustrates some data generation operations. Ideally, the software engineering shall implement a generator that executes these operations. This dataset augmentation specification is used in the next activity to implement the dataset generation functions.

Thanks to the SEMKIS process, we have successfully specified a dataset augmentation based on the specified improved key-properties. We defined precise dataset augmentation operations for retraining and improving the targetNN during the next process iteration.

Table 5.9 Dataset augmentation specification.

Dataset Augmentation-ID	**Operation description**
DSA1	Customer shall add at least 5 images per equivalence class to be added to the testing dataset.
DSA2	Generate 100 random images with shifted left digits.
DSA3	Generate random images for each equivalence class and add them to the training and development dataset to strengthen the targetNN's recognition.

5.2.9 Activity H - Engineering a data synthesizer

In this section, we instantiate activity H (Engineering a synthesizer) of the SEMKIS process, presented in section 3.6.9. During this activity, we use the previously specified dataset augmentation to our synthesizer (see section 3.6.9) .

According to our method, a synthesizer might be a neural network or a classical program. We decided not to design our synthesizer as a neural network like described in Jahic-et-al's paper [3]. We implemented our synthesizer as a classical program in Python in form of several functions permitting the generation of data. We use a classical program for 2 reasons:

1. The dataset's complexity is adequate for generating synthetic data with logical operations.

2. We require logical operations to generate exactly the required synthetic data for augmenting our datasets.

The advantage of a classical program is that we have a full control over the generation of the synthetic images. These functions have been implemented using different libraries such as numpy[51], skimage [161] and bokeh [53]. We use these libraries for managing our data, visualising the data and generating synthetic data. Listing 5.11 shows some samples of our functions implemented in Python. We use them to generate counter state images with/without noise and different brightness levels.

Thanks to the SEMKIS process and the dataset augmentation specification, we have successfully engineered a synthesizer to perform the dataset augmentation.

```python
def generate_concatenated_images(images, labels):
    number_of_refimages = images.shape[0]
    metercounter_images = []
    image_A_classes = []
    image_B_classes = []
    for i in range(number_of_refimages):
        for j in range(number_of_refimages):
            metercounter_images.append(concatenate_images(images[i], images[j]))
            image_A_classes.append(labels[i])
            image_B_classes.append(labels[j])
    image_A_classes = np.array(image_A_classes)
    image_B_classes = np.array(image_B_classes)
    metercounter_images = np.array(metercounter_images)
    equivalence_classes = concatenate_classes(image_A_classes, image_B_classes)
    return metercounter_images, equivalence_classes

def generate_rangeof_shifted_images(images, labels, parameter_range, step, threshold):
    if not assert_parameters(parameter_range):
        print("Parameters are not valid!")
        return
    shifted_ref_images = []
    equivalence_classes = []
    counter = 0
    while counter < len(images):
        current_img = images[counter]
        current_label = labels[counter]
        if counter == len(images) - 1:
            next_img = images[0]
            next_label = labels[0]
        else:
            next_img = images[counter + 1]
            next_label = labels[counter + 1]

        shifted_ref_images.append(current_img)
        equivalence_classes.append(current_label)

        parameters = np.arange(parameter_range[0], parameter_range[1], step)
        for parameter in parameters:
            para_image, para_label = generate_shifted_images(current_img, next_img, current_label, next_label,
                                                             parameter, threshold)
            shifted_ref_images.append(para_image.reshape(310, 165, 1))
            equivalence_classes.append(para_label.reshape(10))
            parameter += step
        counter += 1
    shifted_ref_images = np.array(shifted_ref_images)
    equivalence_classes = np.array(equivalence_classes)
    return shifted_ref_images, equivalence_classes

def generate_shifted_images(current_image, next_image, current_label, next_label,
     parameter, threshold):
    shift = int(height_ref_img * parameter)
    image_zero = np.copy(current_image.reshape(310, 165))
    image_one = np.copy(next_image.reshape(310, 165))
    if parameter <= threshold:
        equivalence_class = current_label
    else:
        equivalence_class = next_label
    image_zero2one = np.concatenate((image_zero[shift:310, :], image_one[0:shift,
         :]), axis=0)
    return image_zero2one.reshape(310, 165, 1), equivalence_class.reshape(10)

def plotbrightness(img, title, brightness_range, brightness_step):
    for parameter in np.arange(brightness_range[0], brightness_range[1],
        brightness_step):
        images = np.copy(img)
        images *= 255
        images = np.where((255 - images) < 100, 255, images + parameter)
        images /= 255
        plt.imshow(images, cmap='gray')
        plt.title("Brightness: " + str(parameter))
        plt.axis("off")
        plt.show()
    return

def plotnoise(img, mode, title):
    if mode is not None:
        gimg = skimage.util.random_noise(img, mode=mode)
        plt.imshow(gimg, cmap='gray')
    else:
        plt.imshow(img, cmap='gray')
    plt.title(title)
    plt.axis("off")
    plt.show()
```

Listing 5.11 Code snippet of some image generator functions in Python.

5.2.10 Activity I - Generating synthetic dataset

In this section, we instantiate activity I (Generate an augmented dataset with classified synthetic data) of the SEMKIS process, presented in section 3.6.10. During this activity, we use the previously engineered synthesizer to generates some synthetic images of counter meter states. Our function takes as input the 10 reference images of digital digits from 0 to 9 to generate the synthetic digits. We use the functions to generate and classify about 500 random training images. Additionally, we add random noise and adjust the brightness of the images. These modifications shall simulate dirty and hardly readable images. In a second step, we analysed these synthetic images and filtered them with respect to our dataset augmentation specification.

1. The images shall be correctly classified.
2. The meter counter states shall be recognizable.
3. The dataset augmentation specification shall be satisfied.

The 100 selected images will then be used and added to the training dataset. Thus, the resulting dataset will be our augmented dataset. In addition to that the customer provided us with additional images to test the retrained targetNN. We use our updated datasets during the next process iteration to train and test our targetNN.

- Initial training dataset: 645 images.
- Augmented training dataset: 745 images.
- Unchanged development dataset: 40 images.
- Updated test dataset: 600 images.

Thanks to this activity, we have been able to generate some synthetic data with our synthesizer. We successfully generated multiple images of meter counter states and changed the intitial datasets. Thanks to the SEMKIS process, we have been able to update the datasets based from the results obtained through analysing the requirements and key-properties specification.

5.3 2nd process iteration: improving our neural network's recognition skills

In the first process iteration, we engineered, trained and tested a targetNN. We discussed the targetNN's recognition skills (key-properties) with the customer. The targetNN hasn't been validated, which is the reason why we considered reengineering our dataset to improve the targetNN's recognition. The last activity of the previous process iteration has been the dataset augmentation and generated a set of synthetic data.

In this section, we instantiate a 2nd iteration of the SEMKIS process as presented in section 3.6.11 on the counter meter case study. We present a summary of the 2nd process iteration, where we try to improve the targetNN's recognition. Our first task consists in augmenting the training dataset with the generated synthetic data. Moreover, we suppose that the customer provided us with additional images permitting us to augment the testing dataset. Thus, we use the updated testing dataset to verify if our targetNN satisfies the customer's requirements.

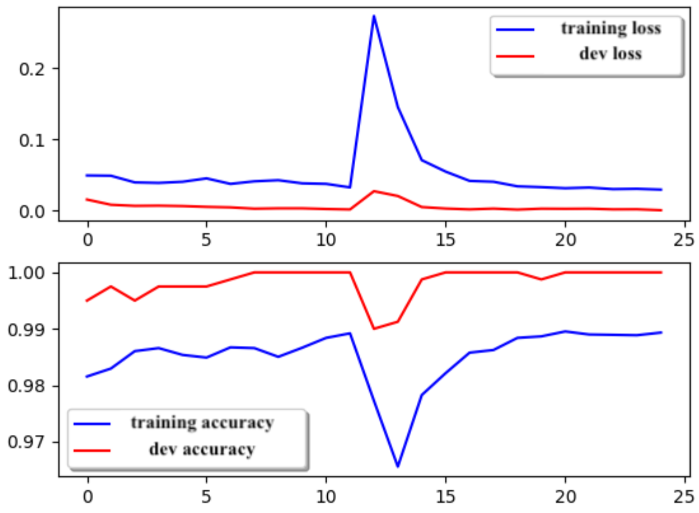


Fig. 5.7 Training diagram of our targetNN v2.0.

We trained our targetNN on the new training dataset and collected the training monitoring data. Thus, we obtained a second version of a targetNN (targetNN v2.0). After the analysis of the train monitoring data, we did not detect any signs of over- and underfitting. Figure 5.7 shows the training diagram after the second training. In the figure, we observe that the targetNN's recognition skills are improving from epoch to epoch. Except in between epoch 10 and 15, the targetNN's recognition skills have dropped by $0,03\%$. The drop of the recognition could happen because of a wrongly targetNN's weights. Since the dataset is shuffled before each epoch, it might happen that we trained the targetNN with a badly ordered dataset. Due to the dataset constellation, it might happen that we had some training anomalies. Nevertheless, the targetNN was still able to improve over time and correcting the training issue.

Afterwards, we categorise and analyse our test monitoring data. First, we identify the key-properties and make first observations on the key-properties. Based on the observations, we specify the key-properties of the retrained targetNN. Listing 5.12 illustrates an updated specification of some quantitative key-properties. We observe that we improved the recognition of the testing dataset by 0.09%. However, we are required to analyse the reason behind the accuracy increase. Analysing further the key-properties, we observe that we improved the targetNN's recognition of the left digit as well as the digits 0 and 9. The only remaining problem is the recognition of the very noisy images (dirty images). Most recognition problems are due to noisy images. Thus, we require to discuss this problem with the customer. Note that a larger quantitative key-properties specification is shown in the appendix under section B.9.

```
    KeyProperties{
        QuantitativeKeyProperties {
            ContinuousKeyProperty ckp3 {
                name "testing dataset accuracy"
                description "Number of images in the used test dataset"
                priority high
                version 1.0
                status toBeDiscussed
                ReachedAccuracy acc_test {accuracy-value 99.59}
            }
            DiscreteKeyProperty dkp5 {
                name "Incorrect recognition of dark or bright images"
                description "NN does not recognise 7 noisy images with dark or bright
                     backgrounds"
                priority high
                version 1.0
                status toBeDiscussed
                incorrectly recognized QuantitativeDataAnalysis c_class for dataset
                    dstest {amount 7}
            }
            DiscreteKeyProperty dkp4 {
                name "Incorrect recognition of digit 9"
                description "NN recognises the right digit on 2 images incorrectly."
                priority high
                version 1.0
                status toBeDiscussed
                incorrectly recognized QuantitativeDataAnalysis c_class for dataset
                    dstest {amount 2 }
            }......
        }}
```

Listing 5.12 Specified quantitative key-properties after 2ndprocess iteration using the SEMKIS-DSL.

Listing 5.13 illustrates an updated specification of some quantitative key-properties. Similar to the quantitative properties, we observe that the overall recognition of the meter counter states has improved a lot. We still have some issues in recognising very noisy images (dirty and bright images). Note that the threshold for selecting the recognised equivalence class has been increased. Thus, we have chosen a more strict selection criteria, which confirms a recognition improvement. Finally, we confirm that the recognition of dirty images shall be discussed with the customer. A larger specification of the quantitative key-properties is shown in the appendix under section B.8.

```
        KeyProperties{
            QualitativeKeyProperties {
                NominalKeyProperty nkp2a {
                    RecognitionAnomaly recoanomly_seven {
                        description "NN recognises the left and right digit on an
                            image very well."
                        equivalenceclasses LeftDigit, RightDigit
                    }}
                NominalKeyProperty nkp2b {
                    RecognitionAnomaly recoanomly_seven {
                        description "NN does not always recognise images of shifted
                            digits on sharp and dirty images"
                        equivalenceclasses LeftDigit, RightDigit
                    }}
                OrdinalKeyProperty okp1 {
                    EquivalenceclassSelection ecSelection {
                        selection-function maximum
                        selection-threshold greaterOrEqualThan 0.9
                    }}
                LogicalKeyProperty lkp2 {
                    EquivalenceClassRecognitionCorrectness ecIncorrectRecoLeftDigit {
                        recognition-correctness incorrectly
                        classification-correctness correctly
                        equivalenceclasses LeftDigit.LeftTwo, LeftDigit.LeftSix,
                            LeftDigit.LeftEight, LeftDigit.LeftNine
                    }}
                ....
            }}
```

Listing 5.13 Specified qualitative key-properties after 2nd process iteration using the SEMKIS-DSL.

Afterwards, we analyse the targetNN's key-properties to define the targetNN's strengths and weaknesses. They are shown in tables 5.10 and 5.11. The main issue of our retrained targetNN is that it does not recognise images with a lot of noise [8]. We present the strengths and weaknesses to the customer. We suppose that the customer claim that the recognition of the images with a lot of noise is not problematic. These meter counters shall be either cleaned or replaced because it is even hard for humans to recognise the meter counter state.

We managed to improve the targetNN's recognition skills. Moreover, the customer is satisfied of the targetNN's recognition. As a result of the discussion with the customer, our retrained targetNN is validated. Our targetNN can be released into production and deployed on the customer's machines.

Nevertheless, it would be possible to improve again the targetNN. Since the targetNN does not recognise dirty images, it is possible to train it to recognize that an image is dirty. Therefore, we shall define a new equivalence classes, called 'dirty images'. The company could be informed to clean or replace a dirty meter counter. This requirement change can be performed in a 3rd process iteration, if the customer wants such feature.

Table 5.10 Identified Weaknesses after 2nd process iteration.

Weakness	**Solution**
Unrecognised very dirty images	Augment the datasets with very dirty images.

Table 5.11 Identified Strengths after 2nd process iteration.

Strengths	**Solution**
Overall recognition correctness.	The targetNN recognised correctly more than 99% of the training, development and testing data.
Specific recognition correctness.	The targetNN recognised very well all kinds of images as described in the requirements.
targetNN's reliability.	The targetNN is not showing signs of over- and underfitting.
targetNN's recognition precision.	The targetNN's recognition precision satisfies the initial requirement.

[8]A lot of noise means that the images are considered to be very dirty and hard to read.

5.4 Results

In this section, we present the results of our case study. We summarise our experimentation, present some achievements and the main results.

In this case study, we conducted an experimentation on the meter counter recognition study. Our aim is to validate our SEMKIS process and the usability of our SEMKIS-DSL by experimenting it on a concrete example. Therefore, we invented a scenario to simulate a real-world problem. The scenario describes a customer that orders a neural network capable of recognising the state of a meter counter. We engineered the neural network by following the activities of our iterative SEMKIS process. In summary, we were able to successfully build a neural network after two process iterations. During the experimentation, we performed different software engineering phases such as requirement engineering, design, implementation and testing.

Our first achievement is the successful execution of the SEMKIS process on a concrete real-world problem. We built a training, development and test dataset required for training and testing a neural network. The datasets consist of synthesized images of meter counter states. We implemented, trained and tested a neural network, which has been designed as convolutional neural network. We analysed the neural network's test results and specified its recognition skills. We successfully specified and performed a dataset augmentation by interpreting the recognition skills. The dataset augmentation allowed us to generate synthetic data and augment our datasets. Thanks to the augmented datasets, we retrain and improved our neural network in the second process iteration.

Our second achievement is the successful support of the SEMKIS-DSL for being capable of engineering datasets to improve its recognition skills. We successfully specified the customer's requirements and the neural network's key-properties with the SEMKIS-DSL. Firstly, we specified the functional and nonfunctional requirements to describe the needs of a customer who orders a neural network. Secondly, the DSL supported us during the specifications to better understand the customer's needs and define precisely the requirements. After the neural network's tests, we successfully specified the neural network's key-properties with the SEMKIS-DSL. The SEMKIS-DSL supported us to ease the comprehension and the description of the neural network's recognition skills. Thanks to these specifications, we were able to compare both specifications in order to verify the neural network's satisfaction of the requirements. Additionally, the specifications serve as a base to define a dataset augmentation for reengineer the datasets and the neural network. Among other, we summarised the main benefits of the SEMKIS-DSL in the following list:

- Verification of the satisfaction of the requirements.

- Guidance for understanding of the neural network's recognition skills.
- Description of the neural network's recognition skills to a customer (often not a NN-expert).
- Base for defining dataset changes to improve the neural network recognition skills.

In summary, we have shown that the SEMKIS process and the SEMKIS-DSL support engineers during their neural network development projects. We showed that the SEMKIS-DSL provides support to lead engineers for rebuilding the datasets to improve neural networks. Moreover, we were able to improve a neural network by following the SEMKIS process and using the SEMKIS-DSL. We reduced the number of incorrectly recognised images and built a neural network that satisfies the customer's requirements. We increased the accuracy (and decreased the loss) on the training, development and testing dataset. The accuracy of the recognition of the testing dataset has been increased from 99.50% to 99.59%. Additionally, we were able to improve the recognition of meter counter states. For example, we improved the recognition of the left digit as well as the right digit. The remaining problem was the recognition of noisy (dirty) and bright images. We simulated that a customer validated our neural network and accepts the issue with the recognition of noisy images. The customer could estimate that this requirement is of less importance since either the meter counter must be cleaned or replaced.

Finally, the results of this case study show that our methodology is promising since we improved the recognition skills of our neural network by augmenting the datasets by considering the key-properties specification. The case study shows a successful application of our method supported by a tool for engineering datasets by considering the customer's requirements in order to improve neural network.

Chapter 6

Perspectives

Abstract

This chapter presents potential improvements and an extension of the SEMKIS methodology. We discuss possible improvements of our methodology as well as potential new research tracks. We present the need for a tool to support the analysis of the training and testing monitoring data, the verification of the SEMKIS usability on a real-world case study, a possible extension of SEMKIS for unsupervised and reinforcement learning problems, advantages of an automated generation of a dataset augmentation specification, and a possible extension of SEMKIS for developing dependable neural networks based on metrics for their resilience.

6.1 Towards a MDE approach supported by a tool

In this section, we present a potential model-driven engineering approach supported by a tool for the analysis of the train and test monitoring data. The motivation behind using a model-driven approach is to maximise the automatisation of the monitoring data analysis to generate a key-properties specification. Our described activities in section 3.6 can be used as input for investigating some further research.

We think that analysing these monitoring data instances is a challenging task for software engineers. Nevertheless, SEMKIS already provides some initial support to ease this task. We presented that software engineers can manually extract relevant information about the training and testing monitoring data to understand the neural network's recognition. However, we think that a model-driven engineering approach supported by a tool can be used to improve the montoring data analysis in order to generate a key-property specification of a neural network. We have analysed the problems faced by software engineers to perform the analysis of some concrete monitoring data instances.

Firstly, the analysis of the recognised data of large datasets might be complicated and time-consuming. This task becomes difficult with an increasing amount and complexity of the data (e.g. videos, images, sound,...) If many data must be manually and visually inspected, it may become difficult to process all data and obtain a good overview of the correctly and incorrectly recognised data. The engineers might often stand in front of a huge amount of shuffled data. This data has to be individually analysed; or some random samples must be analysed, which might lead to an insufficient understanding of the neural network's recognition skills. Thus, it is required to execute more process iterations to improve the neural network until it satisfies the customer's requirements.

Secondly, static confusion matrices contain the amount of correctly and incorrectly recognised equivalence classes. The engineer is supposed to analyse the confusion matrix in order to detect some clusters of incorrectly recognised equivalence classes. This task has to be performed manually, which might be error-prone. The engineer might not detect (or miss) some recognition issues by simply 'watching' at the confusion matrix (especially large matrices).

Thirdly, the accuracy and loss evolution graphs is often analysed only on a static graph. These graphs are visually inspected to verify whether the neural network is over- or underfitting; and has been successfully trained. We think that it might be useful to analyse interactive graphs, which are dynamically updated during the training. Thus, the engineers might faster read the accuracy and loss values with features like hovering. Moreover, engineer might faster detect some potential training issues or problems with the neural network's architecture. These tool features might very useful to visualise more precise information about the loss and accuracy during the neural network's training.

Fourthly, the average accuracy and loss on the different datasets is often only a textual console output. If engineers interpret only this information, they could define wrong conclusions about the neural network's recognition skills. For example, the average accuracy on testing dataset might satisfy the customer's requirements, but does not guarantee that the accuracy on each equivalence class has been satisfied. The average accuracy only provides a general overview of the recognition of an entire dataset, but does not say anything about the recognition of individual and (maybe) highly prioritized equivalence classes (or specific data elements). We think that tools might be useful to link the different monitoring data. This feature would facilitate the analysis of the monitoring data in order to retrieve the optimal information about the recognition skills.

Our iterative process can handle these problems by basically executing it multiple times, until all issues have been detected and removed. Nevertheless, we think that some further research can be investigated to optimise the process execution. This optimization can be achieved with an additional model-driven approach supported by a tool for analysing the monitoring data instances. For all the above described scenarios, it would be very useful to maximise the automatisation of the monitoting data analysis and use model transformation to transform a monitoring data analysis model into our key-properties specification.

The tool supporting this approach might have three types of features:

- Manual features
- Semi-automatic features
- Automatic features

Manual features are any functionalities supporting engineers to manually analyse the monitoring data such as recognised data, the different graphs and the confusion matrices. These features shall help engineers to better visualise and interact with the monitoring data in order to specify the key-properties. The aim of such a feature is to fasten, improve and ease the analysis of the monitoring data. Thus, the engineers should be able to acquire a better understanding of the neural network's recognition skills and use some tool features dedicated to generate appropriate key-properties based on his analysis. Among others, we defined some features that might be useful for engineers:

- **hovering:**: This feature helps to interact with confusion matrices or graphs. More useful information can be included inside the test monitoring data for the engineers. The engineer shall be able to select some options (e.g. insufficient accuracy on an equivalence class, propose equivalence class modifications. . .) to treat the information in the confusion matrices and the graphs.
- **Interacting with testing data:** This feature serves for highlighting the correctly/incorrectly recognised data as well as propose changes on the dataset. Engineers can sort and categorise the equivalence classes and the data based on their recognition correctness. Moreover, they can propose to remove some unrecognised data or move incorrectly classified data to other equivalence classes.

Semi-automatic features are any functionalities that apply a partially automised analysis of the monitoring data for generating structured or new instances of monitoring data. These features shall provide additional information about the neural network's recognition skills to complete the key-properties specification. Similar to the manual features, the goal is to ease the monitoring data analysis and specify a consistent key-properties specification to improve the understanding of the neural network's recogniton skills. Among others, we defined these features that might be useful for engineers:

- **Structured visualisations of the training and testing data:** Depending on the type of data used for training and testing the neural network, it is useful to visualise the data in different ways. Correctly and incorrectly recognised data can be visualised in categories on web pages with a predefined structure. For example, the engineer interacts with the data to organise and structure the data into clusters of incorrectly recognised data. Thus, he would be able to generate key-properties describing insufficient recognition skills for certain sets of data.

- **Automatic evaluation of the confusion matrices:** This feature shall provide additional information about the confusion matrices. Among other types of values, the accuracy and loss values for the different equivalence classes might be automatically computed and visualised with the confusion matrix. Similar to the previous feature, the engineer analyses and interacts with the computed data to identify clusters of incorrectly recognised data. The identified clusters would serve to generate automatically a set of key-properties.

Finally, automatic features are any functionalities that analyse completely the monitoring data and generate information about the neural network's recognition. The aim of these features is to fully automatise the analysis of some monitoring data in order to fasten the activity execution. These feature analyse the monitoring data and generate automatically a key-properties specification.

As a conclusion, we can say that all these features of the tool shall help to provide sufficient support to perform an analysis of the monitoring data following a model-driven engineering approach. The optimal approach would be supported by a tool which automatises completely the analysis of the test monitoring data and generates a key-property specification. The generation feature could be easily extended to generate another customer-friendly recognition skills report written, which describes only on the most important recognition skills (e.g. business critical, requirements changes...) for a customer.

6.2 Industrial assessment of the SEMKIS methodology

In this section, we present a possible assessment of the SEMKIS methodology in an industrial context.

In chapter 5, we presented an academic case study on the recognition of a meter counter [78] recognition problem. The aim of this case study is to experiment and validate our SEMKIS methodology. The results of the experimentation showed that we were able to successfully :

- Specify and update the customer's requirements with the SEMKIS-DSL to engineer appropriate datasets.
- Improve the recognition skills of a neural network with the improved datasets.
- Specify and update the neural network's key-properties with the SEMKIS-DSL to verify the requirements satisfact.
- Show an improvement of the neural network recognition skills by improving its satisfaction of the customer's requirements

We showed that our process is usable and produces an improved neural network that satisfies the requirements. Thanks to the improvement of the recognition skills and the satisfaction of the requirements, we are able to increase the trustworthiness of a neural network. Thanks to the design of SEMKIS-DSL grammar, our requirements and key-property specifications are maintainable as they are required to be updated during each process iteration.

However, there are still some open question that can be adressed as future work. A possible future work would be to perform an industrial assessment of our methodology. The goal would be to assess and validate the SEMKIS methodology by some software engineers. In this assessment, we target to address multiple problems to validate the SEMKIS methodology for industrial usage. It is interesting to compare the convenients and inconvenient of their current approaches versus our methodology. We provide a list of aspects that could be assessed by such an experimentation:

- Reduction of neural network and dataset engineering costs.
- Reduction of dataset and neural network engineering time.
- Evaluation of the learning curve of the process and the SEMKIS-DSL by engineers.
- Determine the complexity of performing the SEMKIS activities and using the SEMKIS-DSL.
- Maintainability: Verify how easilty the process or DSL can be modified and improved in a certain context.
- Readability: Verify the readability of the SEMKIS-DSL specifications.

- Traceability: Verify if neural network's recognition issues can be traced back to some data or requirements.

In order to perform this experimentation, we would invite some deep learning experts from a company to participate in our assessment. We target to perform interviews and a concrete experimentation on the SEMKIS methodology. The assessment would consist of four different phases:

1. **Interview with the software engineers:** The aim of this phase is to discuss the current engineering situation at the company. During this phase, we would like to ask questions like: What are their approaches related to dataset and neural network engineering? What problems do they face during their projects? How do they consider requirements in their deep learning projects? How do they specify and understand the neural network's recognition skills? How do they reengineer datasets? What are the average costs of engineering datasets and neural network using their approaches?...

2. **Experimentation with the SEMKIS methodology:** The aim of this phase is to provide the SEMKIS methodology and SEMKIS-DSL to the engineers to experiment it. The engineers have to work on a concrete project by following the activities described in our project. They shall also use the domain-specific language to specify the requirements and key-properties.

3. **2nd Interview with the software engineers:** The aim of this phase is to interview the software engineers in order to learn more about their understanding of and the usability of SEMKIS. The interviews shall provide sufficient information to verify different properties of the process domain-specific language. Here are a few questions that could be addressed during this phase: How was their general experience to learn with the SEMKIS approach? Have they been able to reduce the engineering costs of your project? Have they been able to reduce the development time? How long did it take to understand SEMKIS and use the DSL? What problems did they encounter during the application of the process and usage of the DSL?...

4. **Evaluate the interviews and experimentations:** The aim of this phase is to analyse the 2 interviews and the experimentation. We target to compare their previous experiences with the SEMKIS methodology. Moreover, we analyse the performed experimentations to learn more about the SEMKIS-DSL as well as the produced datasets and neural networks.

Finally, the assessment shall provide sufficient data to assess the SEMKIS methodology in an industrial context.

6.3 SEMKIS usability on unsupervised and reinforcement learning problems

In this section, we present some potential future work on the usability of the SEMKIS in unsupervised [1] and reinforcement learning[2] problems in deep learning.

In this thesis, we designed our methodology exclusively for supervised learning problems such as the recognition of labelled images. The main aspect in supervised learning is that datasets consist of labelled data. Therefore, we designed our datasets used in our experiments with labelled images. We showed a successful application of our methodology for supervised learning problems.

Nevertheless, a perspective is to study the applicability of other kinds of problems such as unsupervised and reinforcement learning problems. It is worth to perform some research in order to verify whether our methodology is applicable for other learning paradigms. Thus, our methodology could also support software engineers to engineer dataset and improve their neural networks for such kind of problems. The dataset aspects of unsupervised and reinforcement learning that differ from the ones of supervised learning are:

- In unsupervised learning, the datasets consists of unlabeled data.
- In reinforcement learning, the data used for training the neural network consists of information about the state of the environment in which it operates and rewards (or punishments) for its actions taken to change the state.

Concerning reinforcement learning, an additional complexity has to be taken into an account for designing a dataset. Since the neural network interact through action with the environment, the dataset needs to updated in real-time based on the environmental state changes and the reward for the neural network. Thus, it would be required to redefine certain notions of the SEMKIS process in order to be usable in the context of reinforcement learning.

In both context, we think that further research shall be investigated to verify if the SEMKIS process could be used to engineer datasets for improving neural networks. It would be probably required to redefine certain concepts of our process, like for instance different data objects. These data objects shall be verified in the context of these learning paradigms:

[1]Gharamani[39] describes that *"in **unsupervised learning** a machine (e.g. neural network...) simply receives only inputs during its training, but does not obtain any targeted outputs or rewards from its environment. Unsupervised learning can be thought of as finding patterns in the data...*

[2]Gharamani[39] describes that *"in **reinforcement learning** a machine (e.g. neural network...) interacts with its environment by producing different actions. These actions affect the state of the environment, which in turn results in the machine receiving some scalar rewards (or punishments). The goal of the machine is to learn to act in a way that maximises the future rewards (or minimises the punishments) over its lifetime."*

- Raw datasets.
- Equivalence classes.
- Synthesizer.
- Augmented datasets.
- Key-properties specification.

However, we think that our iterative process is sufficiently generalized to map our concepts to these two learning problems. The goal would be the same as for supervised learning problems to iteratively improve the datasets in order to improve datasets. For unsupervised learning problems, the engineer would have to revise and augment the datasets with unlabeled data. For reinforcement learning problems, the engineer would have to reengineer the data collected from the environment to describe the state, that serves as input for the neural network.

Finally, software engineers could profit from our methodology to engineer their neural networks also for unsupervised and reinforcement learning problems and not only for supervised learning problems.

6.4 Automated generation of a dataset augmentation specification.

In this section, we present a potential method for automating the generation of synthetic data based on the specification of the neural network's key-properties[3].

In our SEMKIS process, we defined an activity where neural networks shall be trained on a training datasets. During this activity, the neural network learns to recognise the correct equivalence class for each data element. In a second step, we defined an activity where neural networks shall be tested on the testing dataset. The results of these tests, called test monitoring data, has to be analysed in order to understand the neural network's recognition. Therefore, we introduced the notion of key-properties to specify the neural network's recognition skills using a domain-specific language.

It would be very interesting to perform some further research on automatising the analysis of the key-properties and the generation of the synthetic dataset augmentation specification. The dataset augmentation specification serves as a base to engineer a synthesizer for the

[3]Reminder: The key-properties of a neural network's describe its learned recognition skills during the training.

generation of synthetic data and augmenting the datasets. Looking from a software engineering perspective, we would suggest to use model-driven engineering MDE. Nevertheless, we think that it would be possible to automatise the model transformation of the key-properties specification. We introduced a domain-specific language, called SEMKIS-DSL, for specifying the key-properties. Thus, it would be required to analyse our metamodel to deduce a model transformation to a metamodel of a dataset augmentation specification. Since we implemented the SEMKIS-DSL using Xtext, it would be possible to extend our tool with a dataset augmentation specification generator, implemented in Xtend. The generator would take as input a key-properties specification and output the resulting dataset augentation specification.

Looking at the related work, most papers [162, 91, 94, 89] talking about dataset augmentation specifications present approaches and techniques to perform a dataset augmentation. These dataset augmentation approaches are usually not formally specified. Typically, the data generation is performed without a rigorous process. The engineers only describe the amount of generated synthetic data. In other papers, we found some approaches using AI to specify a dataset augmentation. Cubuk et al. [163] present an AI software, called AutoAugment. The software permits to choose the best possible dataset augmentation based on probability computations. Finally, we can conclude that there is a need for methods and tools to facilitate and automatise the specification of a dataset augmentation. Thus, it is potential future work based on our SEMKIS process and the SEMKIS-DSL.

6.5 Metrics for the resilience of neural networks

In this section, we present a potential extension of the SEMKIS methodology with the DREF framework[164] permitting to engineer dependable neural networks.

Guelfi presents the DREF framework, a formal framework for dependability and resilience from a software engineering perspective, in his paper. He describes the dependability as characteristics of computer system's trustworthiness. The dependability concepts are categorised in:

- **attributes** such as availability, reliability, safety, confidentiality, integrity and maintainability.
- **threats** such as faults, errors, failures.
- **means** such as fault prevention, fault tolerance, fault removal and fault forecasting.

A software system shall satisfy the attributes in order to be considered as dependable. Typically, this means the software can be considered as confident and delivers the expected service. In his paper, Guelfi introduces a concept called satisfiability, which is a measure for evaluating a property (e.g. requirements…) of an entity (e.g. programs, neural networks…). An engineer can define a custom grading scale to specify the satisfiability of a property as real-number. Additionally, he introduces a concept, called tolerance threshold, which describes the acceptable value range of the software's satisfiability. Based on the evolution of the software in time, the changes of the satisfiability and tolerance threshold can be visualised in an evolution graph.

Since we introduced the SEMKIS process and the SEMKIS-DSL, we think that further research shall be investigated on the dependability of neural networks. The DREF framework could be included thanks to the current design of our SEMKIS process and our SEMKIS-DSL. Our domain-specific language offers the specification of the customer's requirements and key-properties of a neural network. These concepts can be used to measure and evaluate the satisfiability of a neural network. The aim would be whether the key-properties satisfy the customer's requirements. Moreover, the tolerance threshold of the satisfiability can be defined together with the customer for each requirement. While engineering the datasets to improve a neural network, the engineer could focus on improving the satisfiability of the requirements. On the other hand, the engineer can update the requirement's specification (and the tolerance threshold), if the customer estimates some requirements are less important. The evolution of the satisfiability can be represented in graphs. The graphs can serve in discussion to validate the neural network or process to the next process iteration.

Chapter 7

Conclusion

In this thesis, we have defined the SEMKIS methodology for dataset engineering to improve neural networks. The SEMKIS methodology consists of these four components:

- The **SEMKIS process** defines an ordered set of activities describing an iterative workflow related to engineering datasets for improving neural networks. It defines different data objects serving as the activity's inputs and outputs. Moreover, it defines human stakeholders responsible for activities' execution. The SEMKIS process is presented in chapter 3 as a business process using the BPMN 2.0 modeling language[21].

- The **SEMKIS subprocess for the analysis of the test monitoring data** defines an ordered set of activities describing a workflow related to analysing and validating the neural network's test results for specifying a dataset augmentation. Sme as the SEMKIS process, it defines different data objects serving as the activity's inputs and outputs as well as responsible for the execution of the different activites. This SEMKIS subprocess is presented in chapter 3 in section 3.6.7 as a business process using BPMN.

- The **SEMKIS-DSL** is a textual domain-specific language for the specification of the neural network's requirements and key-properties. It is designed for software or data engineers for their neural network development projects. Firstly, it defines the concepts for specifying the customer's requirements in the beginning of the SEMKIS process. Secondly, it defines the concepts for specifying its recognition skills as key-properties after the analysis of the neural network's tests. The SEMKIS-DSL have been formalised in UML[107] and implemented using the Xtext framework[123]. The DSL is presented in chapter 4

The main contribution of the SEMKIS methodology is our business process relying on systematically reengineering datasets for improving neural networks. Software engineers are using different specifications to design optimal datasets for training neural networks that satisfy their customer's requirements. Our method is fundamentally based on analysing the neural network's learnd recognition skills after the training for deducing a dataset augmentation in order to reengineer the datasets. Moreover, a novelty is that we use a manual model transformations approach for analysing the requirements and key-properties specification to specify a dataset augmentation.

Another contribution is the usage of a domain-specific language for specifying the neural network's requirements and key-properties. The aim of the domain-specific language is to provide support for software engineers to precisely specify the requirements and key-propertes for evaluating and validating the neural network's learns recognition skills. Depending on the evaluation, the neural network might be considered as valid or the specification is used for specifying a dataset augmentation. The domain-specific language encourages software engineers to precisely define their requirements and understand deeply the neural network's recognition skills to engineer optimal datasets for improving neural networks.

In both case studies (MNIST and counter meter casestudy), we showed a successfull improvement of the neural networks in a concrete scenario. The case studies underlines that our methodology is useful for software engineers to improve neural networks to satisfy the requirements by reengineering datasets.

Lastly, we summarise the main advantages of the SEMKIS methodology as:

- Clear defined process composed of precisly described activities for engineering dataset for improving neural networks.
- Designing the dataset and neural network based on a requirements specification.
- Reengineering datasets based on analysis of the neural networks key-properties specification and the requirements satisfaction.
- Supported by a domain-specific language for specifying the neural network's requirements and the key-properties.
- Usage of synthetic data for reengineering the dataset instead of collecting randomly data.

References

[1] Benjamin Jahić. Semkis : Software engineering methodology for knowledge management of intelligent systems. Technical report, University of Luxembourg, 2018.

[2] Benjamin Jahić, Nicolas Guelfi, and Benoît Ries. Semkis-dsl: a domain-specific language for specifying neural networks' key-properties. Technical report, University of Luxembourg, 2020.

[3] Benjamin Jahić, Nicolas Guelfi, and Benoît Ries. Software engineering for dataset augmentation using generative adversarial networks. In *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*, pages 59–66. IEEE, 2019.

[4] Jahić Benjamin, Guelfi Nicolas, and Ries Benoît. Specifying key-properties to improve the recognition skills of neural networks. In *Proceedings of the 2020 European Symposium on Software Engineering*, pages 60–71, 2020.

[5] P Santhanam, Eitan Farchi, and Victor Pankratius. Engineering reliable deep learning systems. *arXiv preprint arXiv:1910.12582*, 2019.

[6] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300. IEEE, 2019.

[7] Meenu Mary John, Helena Holmström Olsson, and Jan Bosch. Towards mlops: A framework and maturity model. In *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 1–8. IEEE, 2021.

[8] Marc Hesenius, Nils Schwenzfeier, Ole Meyer, Wilhelm Koop, and Volker Gruhn. Towards a software engineering process for developing data-driven applications. In *2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, pages 35–41. IEEE, 2019.

[9] Steven Euijong Whang and Jae-Gil Lee. Data collection and quality challenges for deep learning. *Proceedings of the VLDB Endowment*, 13(12):3429–3432, 2020.

[10] Yuji Roh, Geon Heo, and Steven Euijong Whang. A survey on data collection for machine learning: a big data-ai integration perspective. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[11] Ben Hutchinson, Andrew Smart, Alex Hanna, Emily Denton, Christina Greer, Oddur Kjartansson, Parker Barnes, and Margaret Mitchell. Towards accountability for machine learning datasets: Practices from software engineering and infrastructure. In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, pages 560–575, 2021.

[12] Mona Rahimi, Jin LC Guo, Sahar Kokaly, and Marsha Chechik. Toward requirements specification for machine-learned components. In *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW)*, pages 241–244. IEEE, 2019.

[13] Daniel J Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L Sangiovanni-Vincentelli, and Sanjit A Seshia. Scenic: a language for scenario specification and scene generation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 63–78, 2019.

[14] Stuart Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson, 3rd edition edition, 2016.

[15] AD Dongare, RR Kharde, Amit D Kachare, et al. Introduction to artificial neural network. *International Journal of Engineering and Innovative Technology (IJEIT)*, 2(1):189–194, 2012.

[16] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[17] Görkem Giray. A software engineering perspective on engineering machine learning systems: State of the art and challenges. *Journal of Systems and Software*, 180:111031, 2021.

[18] Tim Menzies. The five laws of se for ai. *IEEE Software*, 37(1):81–85, 2019.

[19] Anders Arpteg, Björn Brinne, Luka Crnkovic-Friis, and Jan Bosch. Software engineering challenges of deep learning. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 50–59. IEEE, 2018.

[20] Anthony Senyard, Edmund Kazmierczak, and Leon Sterling. Software engineering methods for neural networks. In *Tenth Asia-Pacific Software Engineering Conference, 2003.*, pages 468–477. IEEE, 2003.

[21] OMG. Business Process Model and Notation (BPMN), Version 2.0.2, December 2013.

[22] John Haugeland. *Artificial intelligence: the very idea*. Cambridge, MA: MIT Press, 1985.

[23] Eugene Charniak and D McDermott. Introduction to artificial intelligence. addison. *Reading, MA*, 1985.

[24] Richard Bellman. An introduction to artificial intelligence: can computer think? Technical report, 1978.

[25] Patrick Henry Winston. *Artificial intelligence*. Addison-Wesley, 3rd edition edition, 1992.

[26] Ray Kurzweil, Robert Richter, Ray Kurzweil, and Martin L Schneider. *The age of intelligent machines*, volume 580. MIT press Cambridge, 1990.

[27] Robert J. Schalkoff. *Artificial Intelligence: An Engineering Approach*. New York: McGraw-Hill, 1990.

[28] Eliane Rich and Kevin Knight. *Artificial Intelligence*. McGraw-Hill Professional, 2nd edition edition, 1991.

[29] George F. Luger and William A. Stubblefield. *Artificial Intelligence (2nd Ed.): Structures and Strategies for Complex Problem-Solving*. Benjamin-Cummings Publishing Co., Inc., USA, 1993.

[30] Tom M Mitchell. *Machine Learning*, volume 1. Mcgraw-hill science, 1997.

[31] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, 2015.

[32] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[33] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[34] Harsh Kukreja, N Bharath, CS Siddesh, and S Kuldeep. An introduction to artificial neural network. *Int J Adv Res Innov Ideas Educ*, 1:27–30, 2016.

[35] Subana Shanmuganathan. Artificial neural network modelling: An introduction. In *Artificial neural network modelling*, pages 1–14. Springer, 2016.

[36] P Sibi, S Allwyn Jones, and P Siddarth. Analysis of different activation functions using back propagation neural networks. *Journal of theoretical and applied information technology*, 47(3):1264–1268, 2013.

[37] Ruoyu Sun. Optimization for deep learning: theory and algorithms. *arXiv preprint arXiv:1912.08957*, 2019.

[38] Pádraig Cunningham, Matthieu Cord, and Sarah Jane Delany. Supervised learning. In *Machine learning techniques for multimedia*, pages 21–49. Springer, 2008.

[39] Zoubin Ghahramani. Unsupervised learning. In *Summer School on Machine Learning*, pages 72–112. Springer, 2003.

[40] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, and Joelle Pineau. An introduction to deep reinforcement learning. *arXiv preprint arXiv:1811.12560*, 2018.

[41] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[42] Michel F Sanner et al. Python: a programming language for software integration and development. *J Mol Graph Model*, 17(1):57–61, 1999.

[43] Jon Hoffman. *Mastering Swift 5: Deep dive into the latest edition of the Swift programming language*. Packt Publishing, 5 edition, 2019.

[44] Antonio Gulli and Sujit Pal. *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[45] Nikhil Ketkar. Introduction to pytorch. In *Deep learning with python*, pages 195–208. Springer, 2017.

[46] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[47] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[48] Rezaul Karim. *Java Deep Learning Projects: Implement 10 real-world deep learning applications using Deeplearning4j and open source APIs*. Packt Publishing, 2018.

[49] Meints Willem. *Deep Learning with Microsoft Cognitive Toolkit Quick Start Guide: A practical guide to building neural networks using Microsoft's open source deep learning framework*. Packt Publishing, 2019.

[50] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv e-prints*, pages arXiv–1605, 2016.

[51] Wes McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. " O'Reilly Media, Inc.", 2012.

[52] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3):90–95, 2007.

[53] Kevin Jolly. *Hands-on data visualization with Bokeh: Interactive web plotting for Python using Bokeh*. Packt Publishing Ltd, 2018.

[54] Data. *Cambridge Dictionary*. Accessed October 12, 2021 [Online].

[55] Dataset. *Cambridge Dictionary*. Accessed October 12, 2021 [Online].

[56] Terrance DeVries and Graham W Taylor. Dataset augmentation in feature space. *arXiv preprint arXiv:1702.05538*, 2017.

[57] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):1–48, 2019.

[58] Ian Sommerville. *Software engineering 9th Edition*. Addison-Wesley Publishing Company, USA, 9th edition, 2015.

[59] Christos Dimitrakakis and Christian Savu-Krohn. Cost-minimising strategies for data labelling: optimal stopping and active learning. In *International Symposium on Foundations of Information and Knowledge Systems*, pages 96–111. Springer, 2008.

[60] *12207-2017 - ISO/IEC/IEEE International Standard - Systems and software engineering – Software life cycle processes*. IEEE, 2017.

[61] Anwar Ur Rehman, Ahmad Kamran Malik, Basit Raza, and Waqar Ali. A hybrid cnn-lstm model for improving accuracy of movie reviews sentiment analysis. *Multimedia Tools and Applications*, 78(18):26597–26613, 2019.

[62] Aboubakar Nasser Samatin Njikam and Huan Zhao. A novel activation function for multilayer feed-forward neural networks. *Applied Intelligence*, 45(1):75–82, 2016.

[63] Lukas Schott, Jonas Rauber, Matthias Bethge, and Wieland Brendel. Towards the first adversarially robust neural network model on mnist. *arXiv preprint arXiv:1805.09190*, 2018.

[64] Rashad Al-Jawfi. Handwriting arabic character recognition lenet using neural network. *Int. Arab J. Inf. Technol.*, 6(3):304–309, 2009.

[65] Tianmei Guo, Jiwen Dong, Henjian Li, and Yunxing Gao. Simple convolutional neural network on image classification. In *2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA)*, pages 721–724. IEEE, 2017.

[66] Adnan Khashman. Modeling cognitive and emotional processes: A novel neural network architecture. *Neural Networks*, 23(10):1155–1163, 2010.

[67] Ronan Collobert and Jason Weston. Fast semantic extraction using a novel neural network architecture. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 560–567, 2007.

[68] Foutse Khomh, Bram Adams, Jinghui Cheng, Marios Fokaefs, and Giuliano Antoniol. Software engineering for machine-learning applications: The road ahead. *IEEE Software*, 35(5):81–84, 2018.

[69] Anthony Senyard, Philip Dart, and Leon Sterling. Towards the software engineering of neural networks: a maturity model. In *Proceedings 2000 Australian Software Engineering Conference*, pages 45–51. IEEE, 2000.

[70] Mark C Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V Weber. Capability maturity model, version 1.1. *IEEE software*, 10(4):18–27, 1993.

[71] Alice Zheng and Amanda Casari. *Feature engineering for machine learning: principles and techniques for data scientists*. " O'Reilly Media, Inc.", 2018.

[72] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. *Ieee Software*, 33(3):94–100, 2016.

[73] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123:176–189, 2017.

[74] Koichi Hamada, Fuyuki Ishikawa, Satoshi Masuda, Tomoyuki Myojin, Yasuharu Nishi, Hideto Ogawa, Takahiro Toku, Susumu Tokumoto, Kazunori Tsuchiya, Yasuhiro Ujita, et al. Guidelines for quality assurance of machine learning-based artificial intelligence. In *SEKE*, pages 335–341, 2020.

[75] Quanzeng You, Jiebo Luo, Hailin Jin, and Jianchao Yang. Building a large scale dataset for image emotion recognition: The fine print and the benchmark. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[76] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[77] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre Van Schaik. Emnist: Extending mnist to handwritten letters. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2921–2926. IEEE, 2017.

[78] Rayson Laroca, Victor Barroso, Matheus A Diniz, Gabriel R Gonçalves, William R Schwartz, and David Menotti. Convolutional neural networks for automatic meter reading. *Journal of Electronic Imaging*, 28(1):013023, 2019.

[79] Weichao Qiu and Alan Yuille. Unrealcv: Connecting computer vision to unreal engine. In *European Conference on Computer Vision*, pages 909–916. Springer, 2016.

[80] German Ros, Laura Sellart, Joanna Materzynska, David Vazquez, and Antonio M Lopez. The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3234–3243, 2016.

[81] Xingchao Peng, Baochen Sun, Karim Ali, and Kate Saenko. Learning deep object detectors from 3d models. In *Proceedings of the IEEE international conference on computer vision*, pages 1278–1286, 2015.

[82] Nikolaus Mayer, Eddy Ilg, Philip Hausser, Philipp Fischer, Daniel Cremers, Alexey Dosovitskiy, and Thomas Brox. A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4040–4048, 2016.

[83] Ankur Handa, Viorica Patraucean, Vijay Badrinarayanan, Simon Stent, and Roberto Cipolla. Understanding real world indoor scenes with synthetic data. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4077–4085, 2016.

[84] Stephan R Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. Playing for data: Ground truth from computer games. In *European conference on computer vision*, pages 102–118. Springer, 2016.

[85] Matthew Johnson-Roberson, Charles Barto, Rounak Mehta, Sharath Nittur Sridhar, Karl Rosaen, and Ram Vasudevan. Driving in the matrix: Can virtual worlds replace human-generated annotations for real world tasks? *arXiv preprint arXiv:1610.01983*, 2016.

[86] Debidatta Dwibedi, Ishan Misra, and Martial Hebert. Cut, paste and learn: Surprisingly easy synthesis for instance detection. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1301–1310, 2017.

[87] Nikolaus Mayer, Eddy Ilg, Philipp Fischer, Caner Hazirbas, Daniel Cremers, Alexey Dosovitskiy, and Thomas Brox. What makes good synthetic training data for learning disparity and optical flow estimation? *International Journal of Computer Vision*, 126(9):942–960, 2018.

[88] Stefan Hinterstoisser, Vincent Lepetit, Paul Wohlhart, and Kurt Konolige. On pre-trained image features and synthetic images for deep learning. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, pages 0–0, 2018.

[89] Agnieszka Mikołajczyk and Michał Grochowski. Data augmentation for improving deep learning in image classification problem. In *2018 international interdisciplinary PhD workshop (IIPhDW)*, pages 117–122. IEEE, 2018.

[90] Patrice Y Simard, David Steinkraus, John C Platt, et al. Best practices for convolutional neural networks applied to visual document analysis. In *Icdar*, volume 3, 2003.

[91] Sebastien C Wong, Adam Gatt, Victor Stamatescu, and Mark D McDonnell. Understanding data augmentation for classification: when to warp? In *2016 international conference on digital image computing: techniques and applications (DICTA)*, pages 1–6. IEEE, 2016.

[92] Xiaodong Cui, Vaibhava Goel, and Brian Kingsbury. Data augmentation for deep neural network acoustic modeling. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 23(9):1469–1477, 2015.

[93] Hiroshi Inoue. Data augmentation by pairing samples for images classification. *arXiv preprint arXiv:1801.02929*, 2018.

[94] Antreas Antoniou, Amos Storkey, and Harrison Edwards. Data augmentation generative adversarial networks. *arXiv preprint arXiv:1711.04340*, 2017.

[95] Arna Ghosh, Biswarup Bhattacharya, and Somnath Basu Roy Chowdhury. Sad-gan: Synthetic autonomous driving using generative adversarial networks. *arXiv preprint arXiv:1611.08788*, 2016.

[96] Changhee Han, Hideaki Hayashi, Leonardo Rundo, Ryosuke Araki, Wataru Shimoda, Shinichi Muramatsu, Yujiro Furukawa, Giancarlo Mauri, and Hideki Nakayama. Gan-based synthetic brain mr image generation. In *2018 IEEE 15th International Symposium on Biomedical Imaging (ISBI 2018)*, pages 734–738. IEEE, 2018.

[97] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A Bharath. Generative adversarial networks: An overview. *IEEE Signal Processing Magazine*, 35(1):53–65, 2018.

[98] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014.

[99] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: Learning augmentation strategies from data. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 113–123, 2019.

[100] Yazhou Yao, Jian Zhang, Fumin Shen, Li Liu, Fan Zhu, Dongxiang Zhang, and Heng Tao Shen. Towards automatic construction of diverse, high-quality image datasets. *IEEE Transactions on Knowledge and Data Engineering*, 32(6):1199–1211, 2019.

[101] Frédéric Fondement and Raul Silaghi. Defining model driven engineering processes. In *Third International Workshop in Software Model Engineering (WiSME), held at the 7th International Conference on the Unified Modeling Language (UML)*. Citeseer, 2004.

[102] J. Bezivin and O. Gerbe. Towards a precise definition of the omg/mda framework. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 273–280, 2001.

[103] Amal Khalil and J. Dingel. Chapter four - optimizing the symbolic execution of evolving rhapsody statecharts. *Adv. Comput.*, 108:145–281, 2018.

[104] Tomaž Kosar, Pablo E. Martinez Lopez, Pablo A. Barrientos, and Marjan Mernik. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology*, 50(5):390–405, 2008.

[105] Frédéric Fondement. Concrete syntax definition for modeling languages. Technical report, EPFL, 2007.

[106] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[107] OMG UML. Unified modeling language. *Object Management Group*, page 105, 2001.

[108] Fabien Campagne. *The MPS language workbench: volume I*, volume 1. Fabien Campagne, 2014.

[109] Benoît Ries, Alfredo Capozucca, and Nicolas Guelfi. Messir: a text-first dsl-based approach for uml requirements engineering (tool demo). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, pages 103–107, 2018.

[110] Nicolas Guelfi, Benjamin Jahic, and Benoît Ries. Tesma: Requirements and design of a tool for educational programs. *Information*, 8(1):37, 2017.

[111] Bill Venners. The java virtual machine. *Java and the Java virtual machine: definition, verification, validation*, 1998.

[112] Visual Paradigm. Visual paradigm for uml. *Visual Paradigm for UML-UML tool for software application development*, 72, 2013.

[113] Hugo Villamizar, Tatiana Escovedo, and Marcos Kalinowski. Requirements engineering for machine learning: A systematic mapping study. In *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 29–36. IEEE, 2021.

[114] Harshitha Challa, Nan Niu, and Reese Johnson. Faulty requirements made valuable: on the role of data quality in deep learning. In *2020 IEEE Seventh International Workshop on Artificial Intelligence for Requirements Engineering (AIRE)*, pages 61–69. IEEE, 2020.

[115] Khan Mohammad Habibullah and Jennifer Horkoff. Non-functional requirements for machine learning: Understanding current use and challenges in industry. In *2021 IEEE 29th International Requirements Engineering Conference (RE)*, pages 13–23. IEEE, 2021.

[116] Hans-Martin Heyn, Eric Knauss, Amna Pir Muhammad, Olof Eriksson, Jennifer Linder, Padmini Subbiah, Shameer Kumar Pradhan, and Sagar Tungal. Requirement engineering challenges for ai-intense systems development. *arXiv preprint arXiv:2103.10270*, 2021.

[117] Andreas Vogelsang and Markus Borg. Requirements engineering for machine learning: Perspectives from data scientists. In *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW)*, pages 245–251. IEEE, 2019.

[118] Tian Zhao and Xiaobing Huang. Design and implementation of deepdsl: A dsl for deep learning. *Computer Languages, Systems & Structures*, 54:39–70, 2018.

[119] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008.

[120] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. Diesel: Dsl for linear algebra and neural net computations on gpus. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 42–51, 2018.

[121] Artur Podobas, Martin Svedin, Steven WD Chien, Ivy B Peng, Naresh Balaji Ravichandran, Pawel Herman, Anders Lansner, and Stefano Markidis. Streambrain: An hpc dsl for brain-like neural networks on heterogeneous systems.

[122] Vicente García-Díaz, Jordán Pascual Espada, Begoña Cristina Pelayo García Bustelo, and Juan Manuel Cueva Lovelle. Towards a standard-based domain-specific platform to solve machine learning-based problems. *IJIMAI*, 3(5):6–12, 2015.

[123] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[124] Jeff Heaton. Encog: library of interchangeable machine learning models for java and c#. *J. Mach. Learn. Res.*, 16:1243–1247, 2015.

[125] Akshay Agrawal, Akshay Naresh Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, et al. Tensorflow eager: A multi-stage, python-embedded dsl for machine learning. *arXiv preprint arXiv:1903.01855*, 2019.

[126] Arvind K Sujeeth, HyoukJoong Lee, Kevin J Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R Atreya, Martin Odersky, and Kunle Olukotun. Optiml: an implicitly parallel domain-specific language for machine learning. In *ICML*, 2011.

[127] Benoit Ries, Nicolas Guelfi, and Benjamin Jahic. An mde method for improving deep learning dataset requirements engineering using alloy and uml. In *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development*, pages 41–52. SCITEPRESS, 2021.

[128] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, pages 303–314, 2018.

[129] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.

[130] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. In *Conference on robot learning*, pages 1–16. PMLR, 2017.

[131] Ankur Handa, Viorica Pătrăucean, Simon Stent, and Roberto Cipolla. Scenenet: An annotated model generator for indoor scene understanding. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5737–5743. IEEE, 2016.

[132] Paulis Barzdins, Edgars Celms, Janis Barzdins, Audris Kalnins, Arturs Sprogis, Mikus Grasmanis, and Sergejs Rikacovs. Metamodel specialization based dsl for dl lifecycle data management. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 1–2, 2020.

[133] Edgars Celms, Janis Barzdins, Audris Kalnins, Paulis Barzdins, Arturs Sprogis, Mikus Grasmanis, and Sergejs Rikacovs. Dsl approach to deep learning lifecycle data management. *Baltic Journal of Modern Computing*, 8(4):597–617, 2020.

[134] Edgars Celms, Janis Barzdins, Audris Kalnins, Arturs Sprogis, Mikus Grasmanis, Sergejs Rikacovs, and Paulis Barzdins. Towards dsl for dl lifecycle data management. In *International Baltic Conference on Databases and Information Systems*, pages 205–218. Springer, 2020.

[135] Judith Clymo, Haik Manukian, Nathanaël Fijalkow, Adrià Gascón, and Brooks Paige. Data generation for neural programming by example. In *International Conference on Artificial Intelligence and Statistics*, pages 3450–3459. PMLR, 2020.

[136] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[137] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[138] Takafumi Sasakawa, Jinglu Hu, and Kotaro Hirasawa. A brainlike learning system with supervised, unsupervised, and reinforcement learning. *Electrical Engineering in Japan*, 162(1):32–39, 2008.

[139] Nayan B Ruparelia. Software development lifecycle models. *ACM SIGSOFT Software Engineering Notes*, 35(3):8–13, 2010.

[140] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[141] E Roy Davies. *Computer vision: principles, algorithms, applications, learning*. Academic Press, 2017.

[142] Neha Gupta. Artificial neural network. *Network and Complex Systems*, 3(1):24–28, 2013.

[143] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6. IEEE, 2017.

[144] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[145] Mark Summerfield. *Programming in Python 3: a complete introduction to the Python language*. Addison-Wesley Professional, 2010.

[146] Simon Du, Jason Lee, Haochuan Li, Liwei Wang, and Xiyu Zhai. Gradient descent finds global minima of deep neural networks. In *International Conference on Machine Learning*, pages 1675–1685. PMLR, 2019.

[147] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[148] Will Koehrsen. Overfitting vs. underfitting: A complete example. *Towards Data Science*, 2018.

[149] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[150] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[151] Debasish Ghosh. *DSLs in action*. Manning Publications Co., 2010.

[152] Jahic Benjamin. Semkis toolkit. https://github.com/Benji91/lu.uni.lassy.phdthesis.semkis.toolkit.experimentations.git, 2021.

[153] L Vogel. Eclipse ide: Eclipse ide based on eclipse 4.2 and 4.3. *vogella series*, 2013.

[154] Guido Krüger and Thomas Stark. *Handbuch der Java-Programmierung*. Pearson Deutschland GmbH, 2009.

[155] Vincent Massol and Ted Husted. *JUnit in action*. Manning, 2004.

[156] Benjamin Jahić, Nicolas Guelfi, and Benoît Ries. Semkis-dsl: A domain-specific language for improving requirements engineering of neural networks. Technical report, University of Luxembourg, 2020. under review.

[157] Jahic Benjamin. Semkis process - case studies. https://github.com/Benji91/lu.uni.lassy.phdthesis.semkis.experimentations.git, 2020.

[158] Pieter-Tjerk De Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinstein. A tutorial on the cross-entropy method. *Annals of operations research*, 134(1):19–67, 2005.

[159] Usha Ruby and Vamsidhar Yendapalli. Binary cross entropy with deep learning technique for image classification. *Int. J. Adv. Trends Comput. Sci. Eng*, 9(10), 2020.

[160] Quazi Nafiul Islam. *Mastering PyCharm*. Packt Publishing Ltd, 2015.

[161] François Boulogne, Joshua D Warner, and Emmanuelle Neil Yager. Scikit-image: Image processing in python. *J. PeerJ*, 2:453, 2014.

[162] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[163] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: Learning augmentation policies from data. *arXiv preprint arXiv:1805.09501*, 2018.

[164] Nicolas Guelfi. A formal framework for dependability and resilience from a software engineering perspective. *Open Computer Science*, 1(3):294–328, 2011.

Appendix A

SEMKIS - domain-specific language

A.1 SEMKIS-DSL Grammar

```
grammar lu.uni.lassy.phd.dsl.semkis.Semkis with org.eclipse.xtext.common.Terminals

generate semkis 'http://www.uni.lu/lassy/phd/dsl/semkis/Semkis'

Model:
    elements+=(ctRequirements
        | ctDatasets
        | ctEquivalenceClasses
        | ctDataElements
        | ctKeyProperties
    )*;

//*******************************************************************************
// Dataset specification
ctDatasets :
    {ctDatasets} 'Datasets' '{'
        datasets += ctDataset*
    '}';

ctDataset:
    'Dataset' name=ID ('for' 'process-iteration' processiteration=INT)? '{'
        ('name' fullname=STRING)
        ('description' description=STRING)?
        ('dataset-type' type=enDATASETTYPE)?
        ('version' version=REAL)?
        ('size' size=INT)?
        ('format' '['format+=INT('x'format+=INT)*']')?
        ('equivalenceclasses' '{'
            equivalenceclasses+=[ctEquivalenceClass|FQN]('(''numElements' classsize+=
                INT')')?(','equivalenceclasses+=[ctEquivalenceClass|FQN] ('(''
                numElements' classsize+=INT')')?)*
            '}'
        )?
    '}';

//EquivalenceClass specification
ctEquivalenceClasses :
    {ctEquivalenceClasses} 'EquivalenceClasses' '{'
        equivalenceClasses += ctEquivalenceClass*
    '}';

ctEquivalenceClass:
    'EquivalenceClass'name=ID '{'
        ('name' fullname=STRING)
        ('description' description=STRING)?
        ('value' value=STRING)?
        ('acceptance-rule'  symbol=enCOMPARATORS threshold=REAL)?
        ('data-elements' dataelements+=[ctDataElement|FQN](','dataelements+=[
            ctDataElement|FQN])*)?
        rnSubEquivalenceClasses+=ctEquivalenceClass*
    '}';

//DataElements specification
ctDataElements :
    {ctDataElements} 'DataElements' '{'
        dataElement += ctDataElement*
    '}';

//DataElement specification
//A data element can be a single neuron (f.ex age, year, percentage, nationality) or
    a set of neurons (e.g. image, voice)
ctDataElement:
    'DataElement' name=ID '{'
        ('name' fullname=STRING)?
        ('description' description=STRING)?
        (('numerical-elements' '{'
            numericalelements+=ctNumericalElement*
        '}'
        )|
        ('image-elements' '{'
            imageobjects+=ctImageElement*
        '}'
        ))?
    '}';

ctNumericalElement:
    'NumericalElement' name=ID '{'
        ('name' fullname=STRING)?
        ('description' description=STRING)?
        ('value-range' interval=dtRealInterval)?
    '}';

ctImageElement:
    'ImageElement' name=ID '{'
        ('name' fullname=STRING)?
        ('description' description=STRING)?
        ('format' '['format+=INT('x'format+=INT)*']')?
        ('pixel-format' pixelformat=enPIXELFORMAT)?
        ('pixel-value-intervals' intervals+=dtRealInterval(';'intervals+=
            dtRealInterval)*)?
        ('image-content' '{'
            imagecontent+=ctImageContent*
        '}')?
    '}';

ctImageContent:
    'ImageContent' name=ID '{'
        ('name' fullname=STRING)?
        ('description' description=STRING)?
        ('position' enImgContentPosition=enPOSITION)?
        ('pixel-value-intervals' intervals+=dtRealInterval(';'intervals+=
            dtRealInterval)*)?
        ('features' features+=enIMAGEMANIPULATION(','features+=enIMAGEMANIPULATION)*)
            ?
        ('custom-features' customFeatures+=STRING(','customFeatures+=STRING)* )?
    '}';

//****************************************************************************
// Requirements specification
/* Structure for the requirements */
ctRequirements:
    {ctRequirements} 'Requirements' '{'
        functionalRequirements = ctFunctionalRequirements
        nonfunctionalRequirements = ctNonFunctionalRequirements
    '}';

//****************************************************************************
//Functional Requirements
/* Structure the requirements in functional and non-functional requirements */
ctFunctionalRequirements:
    {ctFunctionalRequirements} 'functional' 'Requirements' '{'
        targetNNinput += ctTargetNNInput*
        targetNNoutput += ctTargetNNOutput*
    '}';

ctNonFunctionalRequirements:
    {ctNonFunctionalRequirements} 'nonfunctional' 'Requirements' '{'
        requirements += ctNonFunctionalRequirement*
    '}';

// Functional Requirements Specification Concepts:
ctTargetNNInput:
    {ctTargetNNInput} 'NeuralNetworksInput' '{'
        ('description' description=STRING)?
        ('neurons-size' size=INT)?
        ('neurons-format' '['format+=INT(','format+=INT)*']'('['format+=INT(','format
            +=INT)*']')*)?
        ('input-data' '{'
            inputdata+=ctInputdata*
        '}')?
    '}';

ctInputdata:
    'InputData' name=ID '{'
        ('description' description=STRING)?
        ('neuron-values' interval=dtRealInterval)
        ('data-elements' dataelements+=[ctDataElement|FQN](','dataelements+=[
            ctDataElement|FQN])*)?
    '}';

ctTargetNNOutput:
    {ctTargetNNOutput} 'NeuralNetworksOutput' '{'
        ('neurons-size' size=INT)?
        ('neurons-format' '['format+=INT(','format+=INT)*']'('['format+=INT(','format
            +=INT)*']')*)?
        ('output-value-range' interval=dtRealInterval)?
        ('output-neurons' '['equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*']')?
        ('input-data' inputdata+=[ctInputdata|FQN](','inputdata+=[ctInputdata|FQN])*)
    '}';

//*****************************************************************************
ctNonFunctionalRequirement:
     'Requirement' name=ID '{'
        ('description' description=STRING)?
        ('purpose' purpose=STRING)?
        ('priority' priority=PRIORITY)?

        /* Targeted Performance Specification Global and per equivalence class */
        ((accuracy = ctTargetAccuracy)|
        (loss = ctTargetLoss)|
        (recall = ctTargetRecall)|
        (precision = ctTargetPrecision))?
    '}';


 // Nonfunctional Requirements Specification Concepts:
 ctTargetAccuracy:
    'TargetAccuracy' name=ID ('for' 'dataset' dataset=[ctDataset|FQN])? '{'
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?

        ('min-accuracy' min_acc=REAL)?
        ('max-accuracy' max_acc=REAL)?
        ('target-accuracy' target_acc=REAL)?
    '}';

 ctTargetLoss:
    'TargetLoss' name=ID ('for' 'dataset' dataset=[ctDataset|FQN])? '{'
```

```
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?

        ('min-loss' min_loss=REAL)?
        ('max-loss' max_loss=REAL)?
        ('target-loss' target_loss=REAL)?
    '}';

 ctTargetRecall:
    'TargetRecall' name=ID ('for' 'dataset' dataset=[ctDataset|FQN])? '{'
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?

        ('min-recall' min_recall=REAL)?
        ('max-recall' max_recall=REAL)?
        ('target-recall' target_recall=REAL)?
    '}';

 ctTargetPrecision:
    'TargetPrecision' name=ID ('for' 'dataset' dataset=[ctDataset|FQN])? '{'
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?

        ('min-precision' min_precision=REAL)?
        ('max-precision' max_precision=REAL)?
        ('target-precision' target_precision=REAL)?
    '}';

//*******************************************************************************
// KeyProperties
ctKeyProperties:
    {ctKeyProperties} 'KeyProperties' '{'
        qualKeyproperties += ctQualitativeKeyProperties*
        quanKeyproperties += ctQuantitativeKeyProperties*
    '}';

ctQualitativeKeyProperties:
    {ctQualitativeKeyProperties} 'QualitativeKeyProperties' '{'
        nominalkeyproperties += ctNominalKeyProperty*
        ordinalkeyproperties += ctOrdinalKeyProperty*
        logicalkeyproperties += ctLogicalKeyProperty*
    '}';

ctNominalKeyProperty:
    'NominalKeyProperty' name=ID '{'
        untestedClasses += ctUntestedClasses*
        recognitioncharacteristics += ctRecognitionCharacteristic*
        recoAnamalies += ctRecoAnomaly*
    '}';

ctRecognitionCharacteristic:
    'RecognitionCharacteristic' name=ID '{'
```

```
        ('recognizedAmount' amount=INT)?
        ('recognition-ration' ratio=REAL)?
        ('characteristics' characteristics+=STRING(','characteristics+=STRING)*)?
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?
    '}';

ctUntestedClasses:
    'UntestedClasses' name=ID '{'
        ('description' description=STRING)?
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?
    '}';

ctRecoAnomaly:
    'RecognitionAnomaly' name=ID '{'
        ('description' description=STRING)?
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?
        ('data-elements' dataelements+=[ctDataElement|FQN](','dataelements+=[
            ctDataElement|FQN])*)?
    '}';

ctOrdinalKeyProperty:
    'OrdinalKeyProperty' name=ID '{'
        recognitionevaluations += ctRecognitionEvaluation*
        equivalenceclassselection += ctEquivalenceclassSelection*
    '}';

ctRecognitionEvaluation:
    'RecognitionEvaluation' name=ID '{'
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?
        ('recognition-value' recognitionvalue=enEVALUATION)?
    '}';

ctEquivalenceclassSelection:
    'EquivalenceclassSelection' name=ID '{'
        ('selection-function' selection=enSELECTION)?
        ('selection-threshold'  symbol=enCOMPARATORS threshold=REAL)?
    '}';

ctLogicalKeyProperty:
    'LogicalKeyProperty' name=ID '{'
        ecRecognitioncorrectnesses += ctEquivClassRecoCorrectness*
        dataRecognitionCorrectnesses += ctDataClassRecoCorrectness*
    '}';

ctEquivClassRecoCorrectness:
    'EquivalenceClassRecognitionCorrectness' name=ID '{'
        ('recognition-correctness' recognitioncorrectness = enCORRECTNESS)?
        ('classification-correctness' classificationcorrectness = enCORRECTNESS)?
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?
    '}';

ctDataClassRecoCorrectness:
    'DataElementRecognitionCorrectness' name=ID '{'
        ('recognition-correctness' recognitioncorrectness = enCORRECTNESS)?
        ('classification-correctness' classificationcorrectness = enCORRECTNESS)?
        ('dataElements' dataelements+=[ctDataElement|FQN](','dataelements+=[
            ctDataElement|FQN])*)?
    '}';

ctQuantitativeKeyProperties:
    {ctQuantitativeKeyProperties} 'QuantitativeKeyProperties' '{'
        continuouskeyproperties += ctContinuousKeyProperty*
        discretekeyproperties += ctDiscreteKeyProperty*
    '}';

ctContinuousKeyProperty:
    'ContinuousKeyProperty' name=ID '{'
        ('name' title=STRING)?
        ('description' description=STRING)?
        ('priority' priority=PRIORITY)?
        ('version' version=REAL)?
        ('status' status=STATUS)?
        /* Reached Performance Specification Global and per equivalence class */
        ((accuracy = ctReachedAccuracy)|
        (loss = ctReachedLoss)|
        (recall = ctReachedRecall)|
        (precision = ctReachedPrecision))?
    '}';


 // Nonfunctional Requirements Specification Concepts:
 ctReachedAccuracy:
    'ReachedAccuracy' name=ID ('for' 'dataset' dataset=[ctDataset|FQN])? '{'
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?
        ('accuracy-value' accuracy_value=REAL)?
    '}';

 ctReachedLoss:
    'ReachedLoss' name=ID ('for' 'dataset' dataset=[ctDataset|FQN])? '{'
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?
        ('loss-value' loss_value=REAL)?
    '}' ;

 ctReachedRecall:
    'ReachedRecall' name=ID ('for' 'dataset' dataset=[ctDataset|FQN])? '{'
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?
        ('recall-value' recall_value=REAL)?
    '}';

 ctReachedPrecision:
    'ReachedPrecision' name=ID ('for' 'dataset' dataset=[ctDataset|FQN])? '{'
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?
        ('precision-value' precision_value=REAL)?
    '}';

// Discrete KeyProperties
ctDiscreteKeyProperty:
    'DiscreteKeyProperty' name=ID '{'
        ('name' title=STRING)?
        ('description' description=STRING)?
        ('priority' priority=PRIORITY)?
        ('version' version=REAL)?
        ('status' status=STATUS)?

        /* Reached Performance Specification Global and per equivalence class */
        ((quantitativedataanalysis = ctQuantitativeDataAnalysis) |
        (quantitativedatasetanalysis=ctQuantitativeDatasetAnalysis))?
    '}';

ctQuantitativeDatasetAnalysis:
    'QuantitativeDatasetAnalysis' name=ID ('for' 'dataset' dataset=[ctDataset|FQN])
        ?'{'
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?
        'size' size=INT
    '}';

ctQuantitativeDataAnalysis:
    correctness=enCORRECTNESS (recognition=enRECOGNITION |classification =
        enCLASSIFICATION) 'QuantitativeDataAnalysis' name=ID ('for' 'dataset' dataset
        =[ctDataset|FQN])?'{'
        ('equivalenceclasses' equivalenceclass+=[ctEquivalenceClass|FQN](','
            equivalenceclass+=[ctEquivalenceClass|FQN])*)?
        'amount' amount=INT
    '}';

// Auxiliary Concepts
    dtRealInterval: {dtRealInterval} '['lowerbound=REAL ';' upperbound=REAL']';
    'shift' '('  '['direction+=enMOVEDIRECTION','pixel+=INT']'(',''['direction+=
        enMOVEDIRECTION','pixel+=INT']' )*')' |
    'rotate''('angle=INT ',' 'POS['position+=INT (';' position+=INT)*']' ')' |
    'zoom' '('faktor=REAL',' 'POS['position+=INT (';' position+=INT)*']' ')'  |
    'crop' '(' 'POS['position+=INT (';' position+=INT)*']' ',' 'Rect('pixLength=INT
        ','pixHeigth=INT'))' ;

// Definition of Enumerations
```

```
    enDATASETTYPE: 'training' | 'development' | 'testing' | 'input' | 'synthetic' | '
        other';
    enCOMPARATORS: 'greaterThan' | 'lessThan' | 'lessOrEqualThan' | '
        greaterOrEqualThan' | 'sameAs' | 'differentFrom';
    enPIXELFORMAT: 'rgb' | 'rgba';
    enPOSITION: 'bottom' | 'top' | 'left' | 'right' | 'center' | 'background';
    enMOVEDIRECTION : 'up' | 'down' | 'left' | 'right';
    enIMAGEMANIPULATION:
    enCORRECTNESS: 'correctly' | 'incorrectly';
    enRECOGNITION: 'recognized' | 'unrecognized';
    enCLASSIFICATION: 'classified' | 'unclassified';
    enEVALUATION:'excellent' | 'verywell' | 'well' | 'satisfactory' | 'bad' | '
        verybad' |'unrecognizable';
    enSELECTION:'maximum'| 'minimum';
    STATUS : 'toBeDiscussed' | 'toDo' | 'done' | 'inReview' |  'validated';
    PRIORITY: "high" | "medium" | "low";
    REAL:   (neg ='-')? value = INT (dot='.' r=INT)? (('exp'|'Exp') ('+'|'-')? INT)?;
    FQN: ID ('.' ID)*;
    IMAGEPOSITION: position+=INT (';' position+=INT)*;

    CATEGORY: 'quantitative' | 'qualitative' | 'discrete' | 'continuous' | 'nominal'
        | 'ordinal' | 'logical';
    LOSSFUNCTION: "binary_crossentropy" | "categorical_crossentropy" | "
        sparse_categorical_crossentropy"| "mean_squared_error" | "mean_absolute_error
        ";
    ACTIVATION: "relu" | "sigmoid" | "softmax" | "softplus" | "leakyrelu" ;
    BOOLEAN: "true" | "false";

    LAYERTYPE: 'convolutional' | 'pooling' | 'dropout' | 'normalization' | 'input' |
        'output' | 'hidden' | 'other';
    enARCHITECTURES: 'deepFeedForwardNN'  |  'convolutionalNN' | 'recurrentNN' | '
        LongShortTermMemoryNN' | 'generativeAdversarialNN' | 'other' ;

    TIME: INT(':')INT;
    DATE: INT('.'INT)('.'INT);
    REFERENCE: INT('.'INT)*;
```

Listing A.1 SEMKIS-DSL complete grammar.

Appendix B

Experimentation - Counter Meter Case Study

B.1 SEMKIS-DSL Specifications

B.1.1 Equivalence class Specifications

```
EquivalenceClasses {
EquivalenceClass LeftDigit {
    name "equivalence classes of the left digit"
    EquivalenceClass LeftZero {
        name 'Left-Digit-One-Images'
        value '0'
        acceptance-rule greaterThan 0.95
        data-elements imgDigitZero

    }
    EquivalenceClass LeftOne {
        name 'Left-Digit-One-Images'
        value '1'
        acceptance-rule greaterThan 0.95
        data-elements imgDigitOne
    }
    EquivalenceClass LeftTwo {
        name 'Left-Digit-Two-Images'
        value '2'
        acceptance-rule greaterThan 0.95
        data-elements imgDigitTwo
    }

    EquivalenceClass LeftThree {
        name 'Left-Digit-Three-Images'
        value '3'
        acceptance-rule greaterThan 0.95
```

```
            data-elements imgDigitThree
        }

        EquivalenceClass LeftFour {
            name 'Left-Digit-Four-Images'
            value '4'
            acceptance-rule greaterThan 0.95
            data-elements imgDigitFour
        }

        EquivalenceClass LeftFive {
            name 'Left-Digit-Five-Images'
            value '5'
            acceptance-rule greaterThan 0.95
            data-elements imgDigitFive
        }

        EquivalenceClass LeftSix {
            name 'Left-Digit-Six-Images'
            value '6'
            acceptance-rule greaterThan 0.95
            data-elements imgDigitSix
        }

        EquivalenceClass LeftSeven {
            name 'Left-Digit-Seven-Images'
            value '7'
            acceptance-rule greaterThan 0.95
            data-elements imgDigitSeven
        }

        EquivalenceClass LeftEight {
            name 'Left-Digit-Eight-Images'
            value '8'
            acceptance-rule greaterThan 0.95
            data-elements imgDigitEight
        }

        EquivalenceClass LeftNine {
            name 'Left-Digit-Nine-Images'
            value '9'
            acceptance-rule greaterThan 0.95
            data-elements imgDigitNine
        }
    }

    EquivalenceClass RightDigit{
        name "equivalence classes of the right digit"
        EquivalenceClass RightZero {
            name 'Right-Digit-One-Images'
            value '0'
            acceptance-rule greaterThan 0.95
            data-elements imgDigitZero

        }
        EquivalenceClass RightOne {
            name 'Right-Digit-One-Images'
            value '1'
            acceptance-rule greaterThan 0.95
            data-elements imgDigitOne
        }
        EquivalenceClass RightTwo {
            name 'Right-Digit-Two-Images'
            value '2'
            acceptance-rule greaterThan 0.95
            data-elements imgDigitTwo
        }

        EquivalenceClass RightThree {
            name 'Right-Digit-Three-Images'
            value '3'
            acceptance-rule greaterThan 0.95
            data-elements imgDigitThree
        }

        EquivalenceClass RightFour {
            name 'Right-Digit-Four-Images'
            value '4'
            acceptance-rule greaterThan 0.95
            data-elements imgDigitFour
        }

        EquivalenceClass RightFive {
            name 'Right-Digit-Five-Images'
            value '5'
            acceptance-rule greaterThan 0.95
            data-elements imgDigitFive
        }

        EquivalenceClass RightSix {
            name 'Right-Digit-Six-Images'
            value '6'
            acceptance-rule greaterThan 0.95
            data-elements imgDigitSix
        }

        EquivalenceClass RightSeven {
            name 'Right-Digit-Seven-Images'
            value '7'
            acceptance-rule greaterThan 0.95
            data-elements imgDigitSeven
        }

        EquivalenceClass RightEight {
            name 'Right-Digit-Eight-Images'
            value '8'
            acceptance-rule greaterThan 0.95
            data-elements imgDigitEight
        }

        EquivalenceClass RightNine {
            name 'Right-Digit-Nine-Images'
            value '9'
            acceptance-rule greaterThan 0.95
            data-elements imgDigitNine
        }
    }
}
```

Listing B.1 Equivalence Class Specification

B.1.2 DataElements Specifications

```
    DataElements {
        DataElement imgDigitOne {
            name 'Digit One'
            description 'Image of a 7-segment black "1" on white background'
            image-elements {
                ImageElement img1{
                    format [310 x 165]
                    pixel-format rgb
                    pixel-value-intervals [0;255]
                    image-content {
                        ImageContent img1digit {
                            name 'Digit One'
                            description 'black, 7-segment and digital digit "1"'
                            position center
                            pixel-value-intervals [0;10]
                            features shift([up , 10])
                            custom-features 'black shape'
                        }
                        ImageContent img1background {
                            name 'Background'
                            description 'white background'
                            position background
                            pixel-value-intervals [240;255]
                            custom-features 'white background'
                        }
                    }
                }
            }
        }

        DataElement imgDigitTwo {
```

```
        name 'Digit Two'
        description 'Image of a 7-segment black "2" on white background'
        image-elements {
            ImageElement img2{
                format [310 x 165]
                pixel-format rgb
                pixel-value-intervals [0;255]
                image-content {
                    ImageContent img2digit {
                        name 'Digit Two'
                        description 'black, 7-segment and digital digit "2"'
                        position center
                        pixel-value-intervals [0;10]
                        features shift([up , 10])
                        custom-features 'black shape'
                    }
                    ImageContent img2background {
                        name 'Background'
                        description 'white background'
                        position background
                        pixel-value-intervals [240;255]
                        custom-features 'white background'
                    }
                }
            }
        }
    }

    DataElement imgDigitThree {
        name 'Digit Three'
        description 'Image of a 7-segment black "3" on white background'
        image-elements {
            ImageElement img3{
                format [310 x 165]
                pixel-format rgb
                pixel-value-intervals [0;255]
                image-content {
                    ImageContent img3digit {
                        name 'Digit Three'
                        description 'black, 7-segment and digital digit "3"'
                        position center
                        pixel-value-intervals [0;10]
                        features shift([up , 10])
                        custom-features 'black shape'
                    }
                    ImageContent img2background {
                        name 'Background'
                        description 'white background'
                        position background
                        pixel-value-intervals [240;255]
                        custom-features 'white background'
                    }
```

```
                        }
                    }
                }
            }

            DataElement imgDigitFour {
                name 'Digit Four'
                description 'Image of a 7-segment black "4" on white background'
                image-elements {
                    ImageElement img4{
                        format [310 x 165]
                        pixel-format rgb
                        pixel-value-intervals [0;255]
                        image-content {
                            ImageContent img4digit {
                                name 'Digit Four'
                                description 'black, 7-segment and digital digit "4"'
                                position center
                                pixel-value-intervals [0;10]
                                features shift([up , 10])
                                custom-features 'black shape'
                            }
                            ImageContent img4background {
                                name 'Background'
                                description 'white background'
                                position background
                                pixel-value-intervals [240;255]
                                custom-features 'white background'
                            }
                        }
                    }
                }
            }

            DataElement imgDigitFive {
                name 'Digit Five'
                description 'Image of a 7-segment black "5" on white background'
                image-elements {
                    ImageElement img5{
                        format [310 x 165]
                        pixel-format rgb
                        pixel-value-intervals [0;255]
                        image-content {
                            ImageContent img5digit {
                                name 'Digit Five'
                                description 'black, 7-segment and digital digit "5"'
                                position center
                                pixel-value-intervals [0;10]
                                features shift([up , 10])
                                custom-features 'black shape'
                            }
                            ImageContent img5background {
```

```
                            name 'Background'
                            description 'white background'
                            position background
                            pixel-value-intervals [240;255]
                            custom-features 'white background'
                        }
                    }
                }
            }
        }

        DataElement imgDigitSix {
            name 'Digit Six'
            description 'Image of a 7-segment black "6" on white background'
            image-elements {
                ImageElement img6{
                    format [310 x 165]
                    pixel-format rgb
                    pixel-value-intervals [0;255]
                    image-content {
                        ImageContent img6digit {
                            name 'Digit Six'
                            description 'black, 7-segment and digital digit "6"'
                            position center
                            pixel-value-intervals [0;10]
                            features shift([up , 10])
                            custom-features 'black shape'
                        }
                        ImageContent img6background {
                            name 'Background'
                            description 'white background'
                            position background
                            pixel-value-intervals [240;255]
                            custom-features 'white background'
                        }
                    }
                }
            }
        }

        DataElement imgDigitSeven {
            description 'Image of a 7-segment black "7" on white background'
            image-elements {
                ImageElement img7{
                    format [310 x 165]
                    pixel-format rgb
                    pixel-value-intervals [0;255]
                    image-content {
                        ImageContent img7digit {
                            name 'Seven'
                            description 'black, 7-segment and digital digit "7"'
                            position center
                            pixel-value-intervals [0;10]
                            features shift([up , 10])
                            custom-features 'black shape'
                        }
                        ImageContent img7background {
                            name 'Background'
                            description 'white background'
                            position background
                            pixel-value-intervals [240;255]
                            custom-features 'white background'
                        }
                    }
                }
            }
        }

        DataElement imgDigitEight {
            name 'Digit Eight'
            description 'Image of a 7-segment black "8" on white background'
            image-elements {
                ImageElement img8{
                    format [310 x 165]
                    pixel-format rgb
                    pixel-value-intervals [0;255]
                    image-content {
                        ImageContent img8digit {
                            name 'Digit Eight'
                            description 'black, 7-segment and digital digit "8"'
                            position center
                            pixel-value-intervals [0;10]
                            features shift([up , 10])
                            custom-features 'black shape'
                        }
                        ImageContent img8background {
                            name 'Background'
                            description 'white background'
                            position background
                            pixel-value-intervals [240;255]
                            custom-features 'white background'
                        }
                    }
                }
            }
        }

        DataElement imgDigitNine {
            name 'Digit Nine'
            description 'Image of a 7-segment black "9" on white background'
            image-elements {
                ImageElement img9{
                    format [310 x 165]
                    pixel-format rgb
```

```
                    pixel-value-intervals [0;255]
                    image-content {
                        ImageContent img9digit {
                            name 'Digit Nine'
                            description 'black, 7-segment and digital digit "9"'
                            position center
                            pixel-value-intervals [0;10]
                            features shift([up , 10])
                            custom-features 'black shape'
                        }
                        ImageContent img9background {
                            name 'Background'
                            description 'white background'
                            position background
                            pixel-value-intervals [240;255]
                            custom-features 'white background'
                        }
                    }
                }
            }
        }

        DataElement imgDigitZero {
            name 'Digit Zero'
            description 'Image of a 7-segment black "0" on white background'
            image-elements {
                ImageElement img0{
                    format [310 x 165]
                    pixel-format rgb
                    pixel-value-intervals [0;255]
                    image-content {
                        ImageContent img0digit {
                            name 'Digit Zero'
                            description 'black, 7-segment and digital digit "0"'
                            position center
                            pixel-value-intervals [0;10]
                            features shift([up , 10])
                            custom-features 'black shape'
                        }
                        ImageContent img0background {
                            name 'Background'
                            description 'white background'
                            position background
                            pixel-value-intervals [240;255]
                            custom-features 'white background'
                        }
                    }
                }
            }
        }
    }
```

Listing B.2 Dataelements Specification

B.1.3 Dataset Specifications

```
Datasets {
    Dataset dstrain for process-iteration 0 {
        name 'training dataset'
        description 'The target neural network shall be trained on the training data
            set.'
        dataset-type training
        version 0.1
        size 645
        format [310 x 330 x 645]

        equivalenceclasses{
            LeftDigit.LeftZero      (numElements 60),
            LeftDigit.LeftOne       (numElements 82),
            LeftDigit.LeftTwo       (numElements 54),
            LeftDigit.LeftThree     (numElements 55),
            LeftDigit.LeftFour      (numElements 81),
            LeftDigit.LeftFive      (numElements 49),
            LeftDigit.LeftSix       (numElements 65),
            LeftDigit.LeftSeven     (numElements 65),
            LeftDigit.LeftEight     (numElements 70),
            LeftDigit.LeftNine      (numElements 74),
            RightDigit.RightZero    (numElements 67),
            RightDigit.RightOne     (numElements 46),
            RightDigit.RightTwo     (numElements 59),
            RightDigit.RightThree   (numElements 63),
            RightDigit.RightFour    (numElements 69),
            RightDigit.RightFive    (numElements 56),
            RightDigit.RightSix     (numElements 63),
            RightDigit.RightSeven   (numElements 71),
            RightDigit.RightEight   (numElements 70),
            RightDigit.RightNine    (numElements 81)
        }
    }

    Dataset dstest for process-iteration 0 {
        name 'testing dataset'
        description 'The trained target neural network shall be tested on the testing
             dataset.'
        dataset-type testing
        version 0.1
        size 172
        format [ 310 x 330 x 172]
        equivalenceclasses{
            LeftDigit.LeftZero      (numElements 18),
            LeftDigit.LeftOne       (numElements 10),
            LeftDigit.LeftTwo       (numElements 14),
            LeftDigit.LeftThree     (numElements 18),
            LeftDigit.LeftFour      (numElements 22),
            LeftDigit.LeftFive      (numElements 15),
            LeftDigit.LeftSix       (numElements 19),
            LeftDigit.LeftSeven     (numElements 16),
            LeftDigit.LeftEight     (numElements 17),
            LeftDigit.LeftNine      (numElements 23),
            RightDigit.RightZero    (numElements 20),
            RightDigit.RightOne     (numElements 11),
            RightDigit.RightTwo     (numElements 21),
            RightDigit.RightThree   (numElements 17),
            RightDigit.RightFour    (numElements 8),
            RightDigit.RightFive    (numElements 20),
            RightDigit.RightSix     (numElements 23),
            RightDigit.RightSeven   (numElements 12),
            RightDigit.RightEight   (numElements 16),
            RightDigit.RightNine    (numElements 24)
        }

    }

    Dataset dsdev for process-iteration 0 {
        name 'development dataset'
        description 'The target neural network shall be tested during the training to
             detect recognition issues.'
        dataset-type development
        version 0.1
        size 40
        format [ 310 x 330 x 40]
        equivalenceclasses{
            LeftDigit.LeftZero      (numElements 3),
            LeftDigit.LeftOne       (numElements 4),
            LeftDigit.LeftTwo       (numElements 3),
            LeftDigit.LeftThree     (numElements 4),
            LeftDigit.LeftFour      (numElements 2),
            LeftDigit.LeftFive      (numElements 4),
            LeftDigit.LeftSix       (numElements 5),
            LeftDigit.LeftSeven     (numElements 2),
            LeftDigit.LeftEight     (numElements 4),
            LeftDigit.LeftNine      (numElements 9),
            RightDigit.RightZero    (numElements 8),
            RightDigit.RightOne     (numElements 5),
            RightDigit.RightTwo     (numElements 1),
            RightDigit.RightThree   (numElements 5),
            RightDigit.RightFour    (numElements 4),
            RightDigit.RightFive    (numElements 3),
            RightDigit.RightSix     (numElements 3),
            RightDigit.RightSeven   (numElements 6),
            RightDigit.RightEight   (numElements 4),
            RightDigit.RightNine    (numElements 1)
        }
    }
}
```

Listing B.3 Dataset Specification

B.1.4 Nonfunctional Requirements Specification

```
    nonfunctional Requirements {
        Requirement NFR1 {
            description 'Global accuracy on training dataset'
            purpose 'Defining the tolerated accuracy on the training dataset'
            priority high
            TargetAccuracy tarGlobalTrainAcc for dataset dstrain{
                min-accuracy 99
                max-accuracy 100
                target-accuracy 99.9
            }
        }

        Requirement NFR2 {
            description 'Global accuracy on development dataset'
            purpose 'Defining the tolerated accuracy on the development dataset'
            priority high
            TargetAccuracy tarGlobalDevAcc for dataset dsdev{
                min-accuracy 97
                max-accuracy 100
                target-accuracy 99.5
            }
        }

        Requirement NFR3 {
            description 'Global accuracy on testing dstest'
            purpose 'Defining the tolerated accuracy on the testing dataset'
            priority high
            TargetAccuracy tarGlobalTestAcc for dataset dstest{
                min-accuracy 95
                max-accuracy 100
                target-accuracy 99.7
            }
        }

        Requirement NFR4 {
            description 'Global loss on training dataset'
            purpose 'Defining the tolerated loss on the training dataset'
            priority high
            TargetLoss tarGlobalTrainLoss for dataset dstrain{
                min-loss 0.001
                max-loss 0.02
                target-loss 0.004
            }
        }

        Requirement NFR5 {
            description 'Global loss on development dataset'
            purpose 'Defining the tolerated loss on the development dataset'
            priority high
            TargetLoss tarGlobalDevLoss for dataset dsdev{
                min-loss0.009
                max-loss 0.03
                target-loss 0.03
            }
        }

        Requirement NFR6 {
            description 'Global loss on testing dstest'
            purpose 'Defining the tolerated loss on the testing dataset'
            priority high
            TargetLoss tarGlobalTestLoss for dataset dstest{
                min-loss 0.009
                max-loss 0.03
                target-loss 0.02
            }
        }

        Requirement NFR7 {
            description 'Accuracy on testing dataset for the equivalence class
                LeftZero'
            purpose 'Defining the tolerated accuracy for the equivalence class
                LeftZero from the testing dataset'
            priority high
            TargetAccuracy tarEcLeftZeroAcc for dataset dstest{
                equivalenceclasses LeftDigit.LeftZero
                min-accuracy 95
                max-accuracy 100
                target-accuracy 97
            }
        }

        Requirement NFR8 {
            description 'Accuracy on testing dataset for the equivalence class
                LeftOne'
            purpose 'Defining the tolerated accuracy for the equivalence class
                LeftOne from the testing dataset'
            priority high
            TargetAccuracy tarEcLeftOneAcc for dataset dstest{
                equivalenceclasses LeftDigit.LeftOne
                min-accuracy 95
                max-accuracy 100
                target-accuracy 97
            }
        }

        Requirement NFR8 {
            description 'Accuracy on testing dataset for the equivalence class
                LeftOne'
            purpose 'Defining the tolerated accuracy for the equivalence class
                LeftOne from the training dataset'
            priority high
            TargetAccuracy tarEcLeftOneAcc for dataset dstrain{
                equivalenceclasses LeftDigit.LeftOne
                min-accuracy 99
                max-accuracy 100
                target-accuracy 99.9
            }
        }


    }
```

Listing B.4 Nonfunctional requirements specification

B.1.5 Functional Requirements Specification

```
    functional Requirements {
        NeuralNetworksInput {
            neurons-size 102300
            neurons-format [310,330]
            input-data {
                InputData inputZeroZero{
                    neuron-values [0;255]
                    data-elements imgDigitZero,imgDigitZero
                }
                InputData inputZeroOne{
                    neuron-values [0;255]
                    data-elements imgDigitZero,imgDigitOne
                }
                InputData inputZeroTwo{
                    neuron-values [0;255]
                    data-elements imgDigitZero,imgDigitTwo
                }
                InputData inputZeroThree{
                    neuron-values [0;255]
                    data-elements imgDigitZero,imgDigitThree
                }
                InputData inputZerofour{
                    neuron-values [0;255]
                    data-elements imgDigitZero,imgDigitFour
                }
                InputData inputZeroFive{
                    neuron-values [0;255]
                    data-elements imgDigitZero,imgDigitFive
                }

                InputData inputZeroSix{
                    neuron-values [0;255]
                    data-elements imgDigitZero,imgDigitSix
                }

                InputData inputZeroSeven{
                    neuron-values [0;255]
                    data-elements imgDigitZero,imgDigitSeven
                }
            }
            ....
        }

        NeuralNetworksOutput {
            neurons-size 20
            neurons-format [1,20]
            output-value-range [0;1]
            output-neurons [
                LeftDigit.LeftZero, LeftDigit.LeftOne, LeftDigit.LeftTwo, LeftDigit.
                    LeftThree, LeftDigit.LeftFour,
                LeftDigit.LeftFive, LeftDigit.LeftSix, LeftDigit.LeftSeven,LeftDigit.
                    LeftEight, LeftDigit.LeftNine,
                RightDigit.RightZero, LeftDigit.LeftOne, RightDigit.RightTwo,
                    RightDigit.RightThree, RightDigit.RightFour,
                RightDigit.RightFive, RightDigit.RightSix, RightDigit.RightSeven,
                    RightDigit.RightEight, RightDigit.RightNine
            ]
            input-data inputZeroZero,inputZeroTwo, inputZeroThree, inputZerofour,
                inputZeroFive,inputZeroSix,inputZeroSeven
        }
    }
```

Listing B.5 Functional requirements specification

B.1.6 Quantitative Key-Properties Specification

```
    KeyProperties{
        QuantitativeKeyProperties {
            ContinuousKeyProperty ckp1 {
                name "training dataset accuracy"
                description "Number of images in the used test dataset"
                priority high
                version 1.0
                status toBeDiscussed
                ReachedAccuracy acc_train {
                    accuracy-value 99.97
                }

            }

            ContinuousKeyProperty ckp2 {
                name "development dataset accuracy"
                description "Number of images in the used test dataset"
                priority high
                version 1.0
                status toBeDiscussed
                ReachedAccuracy acc_dev {
                    accuracy-value 99.25
                }

            }

            ContinuousKeyProperty ckp3 {
                name "testing dataset accuracy"
                description "Number of images in the used test dataset"
                priority high
                version 1.0
                status toBeDiscussed
                ReachedAccuracy acc_test {
                    accuracy-value 99.50
                }

            }

            DiscreteKeyProperty dkp1 {
                name "test dataset size"
                description "The test dataset size is 172"
                priority high
                version 1.0
                status toBeDiscussed
                QuantitativeDatasetAnalysis dstest_data for dataset dstest{
                    size 172
                }
            }

            DiscreteKeyProperty dkp2 {
                name "Correct classifications"
                description "172 (all) images have been correctly classified."
                priority high
                version 1.0
                status toBeDiscussed

                correctly classified QuantitativeDataAnalysis c_class for dataset
                    dstest {
                    amount 172
                }
            }

            DiscreteKeyProperty dkp3 {
                name "Incorrect recognition of digit 8"
                description "NN recognises the left digit on 8 images incorrectly."
                priority high
                version 1.0
                status toBeDiscussed

                incorrectly recognized QuantitativeDataAnalysis c_class for dataset
                    dstest {
                    amount 8
                }
            }

            DiscreteKeyProperty dkp4 {
                name "Incorrect recognition of digit 9"
                description "NN recognises the right digit on 3 images incorrectly.."
                priority high
                version 1.0
                status toBeDiscussed

                incorrectly recognized QuantitativeDataAnalysis c_class for dataset
                    dstest {
                    amount 2
                }
            }
        }
    }
```

Listing B.6 Quantitative Key-properties specification

B.1.7 Qualitative Key-Properties Specification

```
KeyProperties{
    QualitativeKeyProperties {
        NominalKeyProperty nkp1 {
                UntestedClasses uts1 {
                    description "These classes have not been tested"
                    equivalenceclasses LeftDigit.LeftThree, RightDigit.RightZero
                }
        }

        NominalKeyProperty nkp2 {
            RecognitionAnomaly recoanomly_seven {
                description "targetNN has issues in recognising shifted digits on
                    sharp and dirty images."
                equivalenceclasses LeftDigit, RightDigit
            }
        }

        NominalKeyProperty np3 {
            RecognitionCharacteristic recoChar {
                recognition-ration 60
                characteristics 'test'
                equivalenceclasses LeftDigit, RightDigit
            }
        }

        OrdinalKeyProperty okp1 {
            EquivalenceclassSelection ecSelection {
                selection-function maximum
                selection-threshold greaterOrEqualThan 0.5
            }
        }

        LogicalKeyProperty lkp1 {
            EquivalenceClassRecognitionCorrectness ecCorrectRecoLeftDigit {
                recognition-correctness correctly
                classification-correctness correctly
                equivalenceclasses LeftDigit.LeftOne, LeftDigit.LeftThree, LeftDigit.LeftSix, LeftDigit.LeftSeven
            }
        }

        LogicalKeyProperty lkp2 {
            EquivalenceClassRecognitionCorrectness ecIncorrectRecoLeftDigit {
                recognition-correctness incorrectly
                classification-correctness correctly
                equivalenceclasses LeftDigit.LeftZero, LeftDigit.LeftTwo, LeftDigit.LeftFour, LeftDigit.LeftFive,LeftDigit.LeftEight,LeftDigit.LeftNine
            }
        }

        LogicalKeyProperty lkp3 {
            EquivalenceClassRecognitionCorrectness ecCorrectRecoRightDigit {
                recognition-correctness correctly
                classification-correctness correctly
                equivalenceclasses RightDigit.RightOne, RightDigit.RightTwo, RightDigit.RightThree, RightDigit.RightFour, RightDigit.RightFive,RightDigit.RightSeven, RightDigit.RightEight,RightDigit.RightNine
            }
        }

        LogicalKeyProperty lkp4 {
            EquivalenceClassRecognitionCorrectness ecIncorrectRecoRightDigit {
                recognition-correctness incorrectly
                classification-correctness correctly
                equivalenceclasses RightDigit.RightOne, RightDigit.RightSix
            }
        }
    }
}
```

Listing B.7 Qualitative Key-properties specification

B.1.8 Qualitative Key-Properties Specification after process iteration 2

```
KeyProperties{
    QualitativeKeyProperties {
        NominalKeyProperty nkp2a {
            RecognitionAnomaly recoanomly_seven {
                description "NN recognises the left and right digit on an image
                    very well."
                equivalenceclasses LeftDigit, RightDigit
            }
        }

        NominalKeyProperty nkp2b {
            RecognitionAnomaly recoanomly_seven {
                description "NN does not always recognise images of shifted
                    digits on sharp and dirty images"
                equivalenceclasses LeftDigit, RightDigit
            }
        }

        NominalKeyProperty np3 {
            RecognitionCharacteristic recoChar {
                recognition-ration 90
                characteristics 'test'
                equivalenceclasses LeftDigit, RightDigit
            }
        }

        NominalKeyProperty np4 {
            RecognitionCharacteristic recoChar {
                recognition-ration 60
                characteristics 'Left Zero not recognised on noisy images.'
                equivalenceclasses LeftDigit.LeftZero
            }
        }

        OrdinalKeyProperty okp1 {
            EquivalenceclassSelection ecSelection {
                selection-function maximum
                selection-threshold greaterOrEqualThan 0.9
            }
        }

        LogicalKeyProperty lkp1 {
            EquivalenceClassRecognitionCorrectness ecCorrectRecoLeftDigit {
                recognition-correctness correctly
                classification-correctness correctly
                equivalenceclasses LeftDigit.LeftZero, LeftDigit.LeftOne,
                    LeftDigit.LeftThree, LeftDigit.LeftFour, LeftDigit.LeftFive,
                    LeftDigit.LeftSeven
            }
        }
```

```

            LogicalKeyProperty lkp2 {
                EquivalenceClassRecognitionCorrectness ecIncorrectRecoLeftDigit {
                    recognition-correctness incorrectly
                    classification-correctness correctly
                    equivalenceclasses LeftDigit.LeftTwo, LeftDigit.LeftSix,
                        LeftDigit.LeftEight, LeftDigit.LeftNine
                }
            }

            LogicalKeyProperty lkp3 {
                EquivalenceClassRecognitionCorrectness ecCorrectRecoRightDigit {
                    recognition-correctness correctly
                    classification-correctness correctly
                    equivalenceclasses RightDigit.RightOne, RightDigit.RightZero,
                        RightDigit.RightThree, RightDigit.RightFour, RightDigit.
                        RightFive,RightDigit.RightSeven, RightDigit.RightEight,
                        RightDigit.RightSix
                }
            }

            LogicalKeyProperty lkp4 {
                EquivalenceClassRecognitionCorrectness ecIncorrectRecoRightDigit {
                    recognition-correctness incorrectly
                    classification-correctness correctly
                    equivalenceclasses RightDigit.RightTwo, RightDigit.RightNine
                }
            }
        }
    }
```

Listing B.8 Qualitative Key-properties specification after process iteration 2

B.1.9 Quantitative Key-Properties Specification after process iteration 2

```
    KeyProperties{
        QuantitativeKeyProperties {
            ContinuousKeyProperty ckp1 {
                name "training dataset accuracy"
                description "Number of images in the used test dataset"
                priority high
                version 1.0
                status toBeDiscussed
                ReachedAccuracy acc_train {
                    accuracy-value 100
                }

            }

            ContinuousKeyProperty ckp2 {
                name "development dataset accuracy"
                description "Number of images in the used test dataset"
                priority high
                version 1.0
                status toBeDiscussed
                ReachedAccuracy acc_dev {
                    accuracy-value 100
                }

            }

            ContinuousKeyProperty ckp3 {
                name "testing dataset accuracy"
                description "Number of images in the used test dataset"
                priority high
                version 1.0
                status toBeDiscussed
                ReachedAccuracy acc_test {
                    accuracy-value 99.59
                }

            }

            ContinuousKeyProperty ckp4 {
                name "testing dataset loss"
                description "Number of images in the used test dataset"
                priority high
                version 1.0
                status toBeDiscussed
                ReachedLoss loss_test {
                    loss-value 0.014919
                }

            }

            DiscreteKeyProperty dkp1 {
                name "test dataset size"
                description "The test dataset size is 600"
                priority high
                version 1.0
                status toBeDiscussed
                QuantitativeDatasetAnalysis dstest_data for dataset dstest{
                    size 600
                }
            }


            DiscreteKeyProperty dkp2 {
                name "Correct classifications"
                description "600 (all) images have been correctly classified."
                priority high
                version 1.0
                status toBeDiscussed

                correctly classified QuantitativeDataAnalysis c_class for dataset
                    dstest {
                    amount 600
                }
            }

            DiscreteKeyProperty dkp3 {
                name "Incorrect recognition of digit 0"
                description "NN recognises the left digit on 8 images incorrectly."
                priority high
                status toBeDiscussed
                incorrectly recognized QuantitativeDataAnalysis c_class for dataset
                    dstest {
                    amount 8
                }
            }

            DiscreteKeyProperty dkp4 {
                name "Incorrect recognition of digit 9"
                description "NN recognises the right digit on 2 images incorrectly."
                priority high
                version 1.0
                status toBeDiscussed

                incorrectly recognized QuantitativeDataAnalysis c_class for dataset
                    dstest {
                    amount 2
                }
            }

            DiscreteKeyProperty dkp5 {
                name "Incorrect recognition of dark or bright images"
                description "NN does not recognise 7 noisy images with dark or bright
                     backgrounds"
                priority high
                version 1.0
                status toBeDiscussed

                incorrectly recognized QuantitativeDataAnalysis c_class for dataset
                    dstest {
                    amount 7
                }
            }

            DiscreteKeyProperty dkp6 {
                name "Incorrect recognition of dark digits on a dark background"
                description "NN does not recognise 3 noisy images with dark
                    background and dark digits"
                priority high
```

```
                version 1.0
                status toBeDiscussed

                incorrectly recognized QuantitativeDataAnalysis c_class for dataset
                    dstest {
                    amount 3
                }
            }

            DiscreteKeyProperty dkp7 {
                name "Incorrect recognition of bright digits on a bright background"
                description "NN does not recognise 2 noisy images with bright
                    background and bright digits"
                priority high
                version 1.0
                status toBeDiscussed

                incorrectly recognized QuantitativeDataAnalysis c_class for dataset
                    dstest {
                    amount 2
                }
            }

            DiscreteKeyProperty dkp8 {
                name "Incorrect recognition of dark digits on a bright background"
                description "NN does not recognise 2 noisy images with bright
                    background and dark digits"
                priority high
                version 1.0
                status toBeDiscussed

                incorrectly recognized QuantitativeDataAnalysis c_class for dataset
                    dstest {
                    amount 2
                }
            }
        }
    }
```

Listing B.9 Quantitative Key-properties specification after process iterations 2