LAPPEENRANTA-LAHTI UNIVERSITY OF TECHNOLOGY LUT
School of Engineering Science
Software Engineering

Joonas Virmajoki

DETECTING CODE SMELLS USING ARTIFICIAL INTELLIGENCE – A PROTOTYPE

Examiners: Associate Professor Jussi Kasurinen
Assistant Professor Antti Knutas

TIIVISTELMÄ

Lappeenrannan-Lahden teknillinen yliopisto LUT
School of Engineering Science
Tietotekniikan koulutusohjelma
Joonas Virmajoki

Koodihajujen havaitseminen käyttäen tekoälyä - prototyyppi
Diplomityö 2020

81 sivua, 20 kuvaa, 7 taulukkoa, 10, katkelmaa, 1 liite

Työn tarkastajat: Apulaisprofessori Jussi Kasurinen
Apulaisprofessori Antti Knutas

Hakusanat: koodihaju, tekoäly, koneoppiminen, syväoppiminen, prototyyppi, refaktorointi, neuroverkot.
Keywords: code smell, artificial intelligence, machine learning, deep learning, prototype, refactoring, artificial neural networks.

Tekoäly on yksi aikamme merkittävistä hienouksista. Tekoälyä hyödynnetään ohjelmistoprojektien laadun parantamisessa ja myös itse sovelluksissa. Koodihajut ovat piirteitä lähdekoodissa, jotka indikoivat syvempää ongelmaa, ja ne ovat koodaajien pitkäaikainen riesa. Koodihajut vaikeuttavat ohjelmien ketterää ylläpidettävyyttä, uudelleenkäyttöä ja laajennettavuutta. Lähdekoodia refaktoroimalla voi päästä koodihajuista eroon, mutta ensiksi koodihajut täytyy löytää. Tutkimuksessa tehtiin prototyyppi koodihajujen havaitsemiseen sekä esiteltiin sen suunnittelu ja kehitys. Prototyyppi toteutettiin Python-ohjelmointikielellä, käyttäen koneoppimista, neuroverkkoja ja syväoppimista. Opetus- ja testidata otettiin MLCQ-koodihajuaineistosta, sekä lisäksi dataa kerättiin GitHubin avoimen lähdekoodin Java-kielen ohjelmavarastoista. Prototyyppi onnistui havaitsemaan onnistuneesti ”long method” ja ”feature envy” koodihajuja, vaikka dataa kerättiin ja käytettiin vain suhteellisen vähän prototyypin opettamiseen.

ABSTRACT

Lappeenranta-Lahti University of Technology
School of Engineering Science
Software Engineering
Joonas Virmajoki

Detecting code smells using artificial intelligence – a prototype
Master's Thesis 2020

81 pages, 20 figures, 7 tables, 10 listings, 1 appendix

Examiners: Associate Professor Jussi Kasurinen
Assistant Professor Antti Knutas

Keywords: code smell, artificial intelligence, machine learning, deep learning, prototype, refactoring, artificial neural networks.

Artificial intelligence is one of the major subtleties of our time. Artificial intelligence is utilized in improving the quality of software projects and in applications themselves. Code smells are characteristics in the source code that indicate there is a deeper problem, and they are a long-term nuisance for developers. Code smells make it hard to maintain, reuse, and expand software. You can refactor your source code to get rid of code smells, but first you need to find code smells. In this thesis, I made a prototype for detecting code smells as well as presented its design and development. The prototype was implemented in the Python programming language, using machine learning, neural networks, and deep learning. Training and testing data were taken from the MLCQ code smell dataset, and non-smelly samples were collected from GitHub's open source Java repositories. The prototype was able to detect "long method" and "feature envy" code smells successfully, although only a relatively small amount of data was collected and used for the training of the prototype.

ACKNOWLEDGEMENTS

Thank you so much to my family and friends for all the support during this thesis. I am deeply grateful. Additionally, thank you to Jussi Kasurinen for an interesting topic and guiding me in the right direction.

Joonas Virmajoki

TABLE OF CONTENTS

1 INTRODUCTION 5

1.1 BACKGROUND 5
1.2 GOALS AND DELIMITATIONS 7
1.3 RESEARCH METHOD 8
1.4 STRUCTURE OF THE THESIS 9

2 LITERATURE REVIEW 10

2.1 ARTIFICIAL INTELLIGENCE 10
2.1.1 Concept of artificial intelligence 10
2.1.2 Categorization of artificial intelligence 12
2.1.3 The role of artificial intelligence in software engineering 12
2.1.4 Programming languages for artificial intelligence 14
2.2 MACHINE LEARNING 15
2.2.1 Supervised learning 16
2.2.2 Unsupervised learning 18
2.2.3 Reinforcement learning 19
2.3 DEEP LEARNING 20
2.3.1 Artificial neural network 21
2.3.2 Convolutional neural network 22
2.3.3 Recurrent neural network 23
2.3.4 Summary and comparison 23
2.4 REFACTORING 25
2.4.1 Reasons for refactoring 25
2.5 CODE SMELLS 26
2.5.1 Types of code smells 27
2.5.2 Negative effects and concerns 30
2.5.3 Detection tools 31
2.6 NATURAL LANGUAGE PROCESSING 33
2.6.1 Differences between text and code 33
2.6.2 Representing code 34

2.6.3 Encoding text *35*

3 PROTOTYPE DESIGN **38**

3.1 VISION OF THE SOLUTION 38

3.1.1 Vision statement *39*

3.1.2 Major features *40*

3.1.3 Assumptions and dependencies *40*

3.2 SCOPE AND LIMITATIONS 42

3.2.1 Scope of initial release *42*

3.2.2 Scope of subsequent releases *42*

3.2.3 Limitations and exclusions *42*

3.3 GATHERING AND PREPARATION OF DATA 43

3.3.1 Gathering *43*

3.3.2 Preparation *44*

3.3.3 Structure *45*

3.4 REQUIREMENTS 46

3.4.1 Functional requirements *46*

3.4.2 Non-functional requirements *47*

3.4.3 Use cases *48*

4 PROTOTYPE DEVELOPMENT **49**

4.1 INFRASTRUCTURE 49

4.2 IMPORTING LIBRARIES 51

4.3 LOADING, SHUFFLING AND PREPROCESSING DATA 51

4.4 TOKENIZING DATA 52

4.5 SPLITTING DATA 53

4.6 ADDING PADDING TO DATA 53

4.7 CREATING A MODEL 54

4.8 TRAINING A MODEL 55

4.9 SAVING A MODEL 56

4.10 MAKING A PREDICTION 57

5 RESULTS **58**

5.1 "LONG METHOD" CODE SMELL 59

5.1.1 Training and validation accuracy *59*
5.1.2 Training and validation loss *59*
5.2 "FEATURE ENVY" CODE SMELL 61
5.2.1 Training and validation accuracy *61*
5.2.2 Training and validation loss *62*
6 THREATS TO VALIDITY 63
6.1 CONSTRUCT VALIDITY 63
6.2 INTERNAL VALIDITY 63
6.3 EXTERNAL VALIDITY 63
6.4 RELIABILITY 64
7 CONCLUSIONS AND FUTURE DIRECTIONS 65
REFERENCES 66
APPENDIX

LIST OF SYMBOLS AND ABBREVIATIONS

ACM	Association for Computing Machinery
AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
AST	Abstract Syntax Tree
CNN	Convolutional Neural Network
CSV	Comma-Separated Values
GPU	Graphics Processing Unit
GUI	Graphical User Interface
IBM	International Business Machines Corporation
IEEE	Institute of Electrical and Electronics Engineers
LSTM	Long Short-Term Memory
ML	Machine Learning
MOOC	Massive Open Online Course
NLP	Natural Language Processing
Regex	Regular expression
RNN	Recurrent Neural Network
TF-IDF	Term Frequency-Inverse Document Frequency
TPU	Tensor Processing Unit

1 INTRODUCTION

1.1 Background

Artificial intelligence (AI) is one of the most exciting advances of our time. AI makes it possible that cars can drive their own and intelligent computers can beat humans in strategic games, like chess. AI based recommender systems can predict our music and movie taste with high accuracy. Availability of data and cheaper computing are reinforcing the importance of AI. (Panesar, 2019) The AI field is undergoing enormous growth and as AI become more accessible, its use can be expected to increase in modern software systems (Feldt, et al., 2018). Figure 1 presents how widely the AI field has expanded.

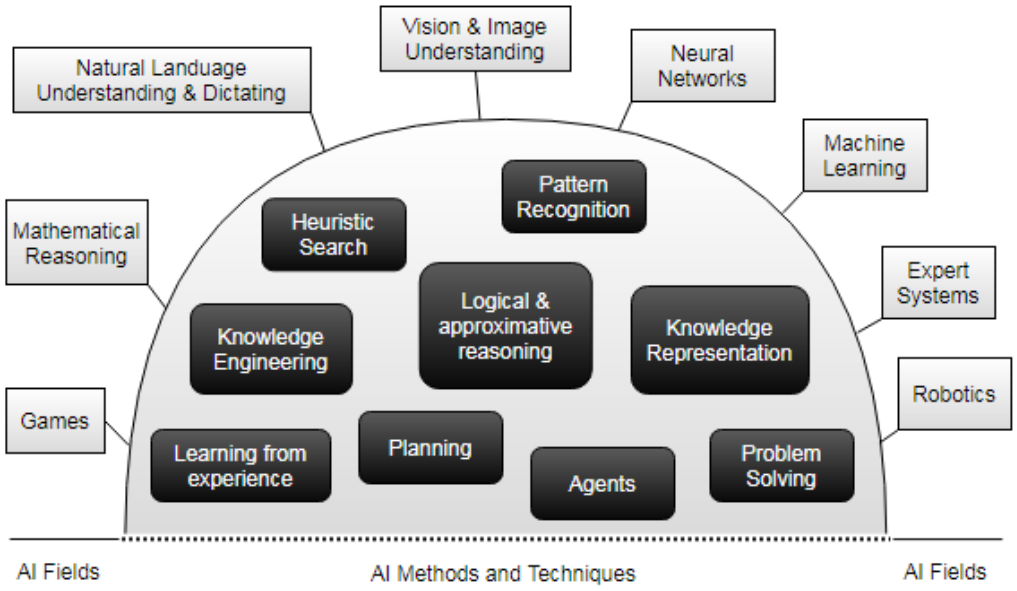

Figure 1. AI fields, methods, and techniques (adapted from Rech and Althoff, 2004)

Software engineering is continuously evolving. Research on the use of AI in software engineering has grown enormously over the last two decades. The quality of a product can be increased by using AI techniques in software development and in software itself. AI has been studied to assist programmers' productivity and program reliability. AI can process a large amount of data and make more accurate predictions than humanly possible. AI technologies are increasingly componentized and can be more easily used and reused, even by beginners. (Ammar, et al., 2012; Feldt, et al., 2018)

Software maintenance is a key part of the software lifecycle and its costs have been continuously growing. Researchers have estimated that even 90 percent of software lifetime cost is related to the maintenance phase. Software maintenance will lead to longer life of software by preventing software aging. Incomplete documentation and low maintenance are factors to increase the costs because defects make it more difficult to expand software. (Dehaghani and Hajrahimi, 2013) We can improve internal software qualities such as reusability, maintainability, and extensibility through refactoring. Refactoring is a process that does not add new features, it just makes the system easier to maintain in the future by improving its internal structure. (Szőke at el., 2015) Jim Highsmith, one of the agile manifesto creators, describes the importance of refactoring as follows (2002):

"Refactoring may be the single most important technical factor in achieving agility."

Technical debt metaphor means the cost of rework what is caused by delivering software fast, but not taking the better approach that would take longer. One symptom of technical debt is code smells, which are indications of poor design and implementation choices. (Szőke at el., 2015) Figure 2 shows a typical refactoring flow, where testing is an essential part of refactoring, ensuring that everything works after the changes. Developers biggest fear is to break software when performing refactoring (Tufano et al., 2017).

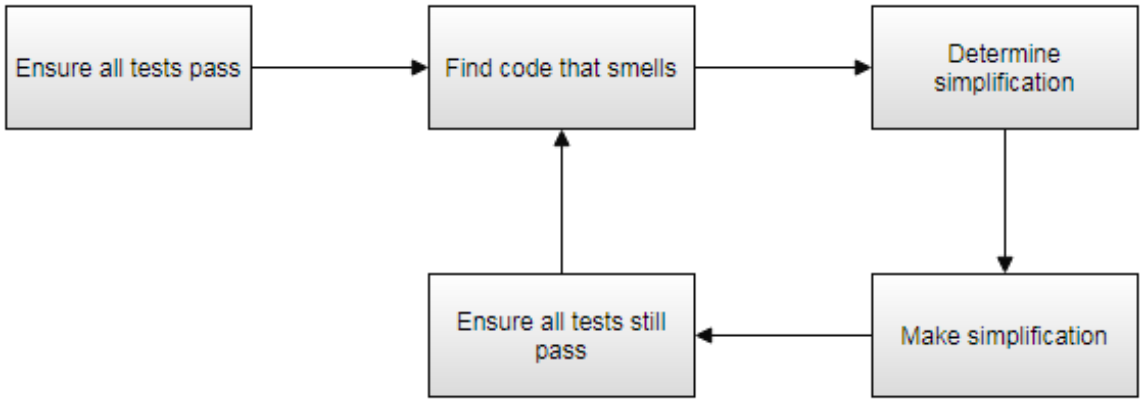

Figure 2. Refactoring flow (adapted from Kasurinen, 2020)

According to Tufano et al. (2017), most of the smell instances are introduced when the files are created. Code smells are generally introduced by developers when adding or editing

existing features, typically close to a deadline. Developers who introduce smells are generally owners of the file. High workload developers tend to be more prone to introducing code smells than others. Most code smells are never removed from system which leads to high survivability of code smells. (Tufano et al., 2017) The earlier code smells are found, the less costs there will be and better the software quality will be (Hadj-Kacem and Bouassida, 2018).

This thesis concentrates on the part of finding code smells using AI techniques. Aim is to design and develop a rudimentary working prototype. Prototypes are usually quick to make, and they allow to evaluate developers' proposal for the design. This thesis explores ideas, different techniques, code smells and selects the appropriate tools for implementation. Yet many of the AI technologies remain only use of the researchers, and there is not as much impact on the software engineering processes and tools. There is still huge gap between research and practice of applying AI to software engineering. (Ammar, et al., 2012)

1.2 Goals and delimitations

The goal of this thesis is to show that I am capable to design and develop a prototype. It does not need to find all code smells or to be perfect. The thesis is not trying to 'prototype' a final product, system, or service, it is more like a research product. The prototype is done for demonstration purposes, idea generation, and new insights. This research is limited to only finding code smells. Thus, it is not expected to tell how to refactor source code or refactor it automatically. In addition, the research is limited to cover only one programming language, and the aim is to find well at least one type of code smell in that certain programming language. To keep it simple, the prototype can only take one method as input and classify if certain code sample smells or not. The prototype is focusing weak AI, which is only good at doing one task well.

The aim of the research is to find answers to the following research questions:

- How code smells are detected and defined?
- How to design and implement a simple AI-based code smell analyzer?
- How well the prototype can perform?

1.3 Research method

The research methodology of this thesis consists of two parts: a literature review and the design research. The literature review was done to help me choose what to build and to get a basic understanding from the topic. It is also important to find out what researchers have done in recent studies. Based on previous good experiences, I selected to use the following popular scholarly literature digital libraries:

- IEEE (Institute of Electrical and Electronics Engineers) Xplore
- LUT Finna
- Google Scholar
- ACM (Association for Computing Machinery) Digital Library
- Springer Link

I used the following search words and their combinations to get suitable results: “AI”, “machine learning”, “code smell”, “software engineering”, “deep learning”, “refactoring” and “classification”. From the search results, I favored the most up-to-date and recent publications, because the field is evolving rapidly. I also used many other online sources. For example, there are plenty of Massive Open Online Courses (MOOCs) dealing with AI.

A design science research methodology can be concisely summarized as: “build to learn”. It creates and evaluates an information technology artifact to solve problems. The design science research process model consists of five steps, as you can see in Figure 3. The first activity is to determine the research problem to motivate the audience and the researcher to pursue solution. Second activity is to define the objectives for a solution. It determines what is the goal and what is possible and feasible to do. Third step is design and development which determines designed functionality and its architecture. The prototype is developed in this phase. The next step is the evaluation of the developed artifact. More precisely, it observes and measure how well the artifact performs using relevant metrics and analysis techniques. Final step is communication, which means spreading the resulting knowledge through a scholar publication. (Peppers, et al., 2007) This thesis is using a problem-centered approach, starting from activity one through to activity five.

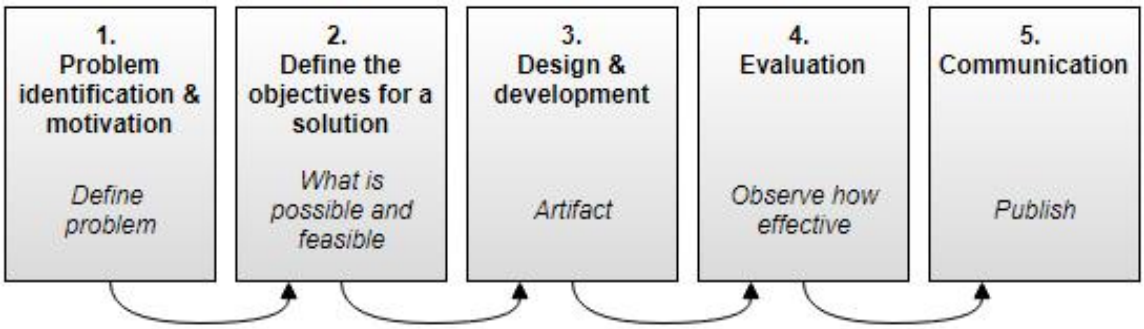

Figure 3. Design science research process model (adapted from Peppers, et al., 2007)

1.4 Structure of the thesis

The first chapter is the introduction to this thesis, and the background to the thesis is also presented. Furthermore, the research questions and research methods are specified. The rest of the thesis is organized as follows. The literature review of this thesis is covered in Chapter 2. The design of the prototype is considered in Chapter 3. The implementation of the prototype is presented in Chapter 4. Results from the prototype are analyzed and evaluated in Chapter 5. Threads to validity are considered in Chapter 6. In other words, how reliable the results are. Finally, conclusions and future directions of the thesis are drawn in Chapter 7.

2 LITERATURE REVIEW

In this chapter, a literature review is presented. The review starts from the introduction to AI and then continues to machine learning (ML). Secondly, refactoring and code smells are presented in more detail. Further, their mutual relation is also described. Finally, natural language processing (NLP) is introduced to show how text can be encoded to ML algorithms.

2.1 Artificial intelligence

This subchapter presents what is AI really and how it can be defined and categorized. The relevance of AI in software engineering is explored and the most common programming languages for AI are presented.

2.1.1 Concept of artificial intelligence

It is difficult to define AI simply and robustly. There is no exact definition of AI, even by AI researchers. The field of AI is constantly redefined. More new topics emerge, and some topics are classified as non-AI. For example, fifty years ago, automatic methods for search and planning were considered to belong to the domain of AI. When methods are becoming well understood, they are likely to be moved from AI to statistics or probability. (Elements of AI, 2019)

In 1950, Alan Turing published *Computing Machinery and Intelligence* and introduced a practical test for computer intelligence, which is known as the Turing test. The Turing test evaluates whether the behavior of a machine is distinguishable from human behavior. (Panesar, 2019) The term artificial intelligence was first defined in 1955 by John McCarthy (Ertel, 2011): “The goal of AI is to develop machines that behave as though they were intelligent.” It was based on the idea that any feature of intelligence can be so precisely described that machine can be made to imitate it (Panesar, 2019). International Business Machines Corporation (IBM) defines AI as everything that makes machines act more intelligently. IBM believe that AI should not replace human but instead extend human capabilities and help to do tasks that human or machines could not do their own. (IBM, 2019)

AI is fundamentally mostly programming. As shown in Figure 4, AI is a subset of computer science. The rising popularity of AI is due to explosion of data through devices and cheaper computing power. IBM estimated that 90% of global data have been created in the last two years. This exponentially generated data allows everything to become smart. More data means more capable of learning, which allows higher accuracy. Data mining is a process to turn raw data to useful information so that machine learning models can learn from existing data. (Panesar, 2019)

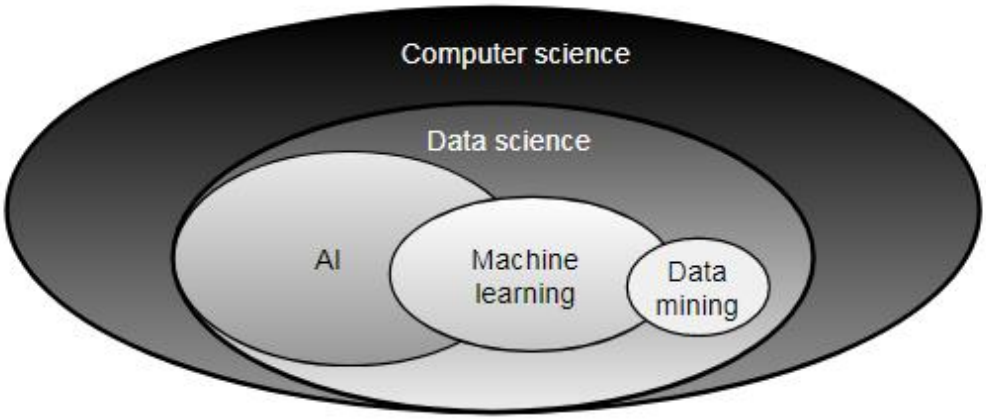

Figure 4. Place of AI in computer science (adapted from Panesar, 2019)

2.1.2 Categorization of artificial intelligence

Table 1 presents how IBM breaks down AI to the three different categories based on machines capability (2019).

Table 1. Categories of AI (adapted from IBM, 2019)

Weak AI	**Strong AI**	**Super AI**
"Weak or Narrow AI is AI that is applied to a specific domain. For example, language translators, virtual assistants, self-driving cars, AI-powered web searches, recommendation engines, and intelligent spam filters. Applied AI can perform specific tasks, but not learn new ones, making decisions based on programmed algorithms, and training data." (IBM, 2019)	"Strong AI or Generalized AI is AI that can interact and operate a wide variety of independent and unrelated tasks. It can learn new tasks to solve new problems, and it does this by teaching itself new strategies. Strong Intelligence is the combination of many AI strategies that learn from experience and can perform at a human level of intelligence." (IBM, 2019)	"Super AI or Conscious AI is AI with human-level consciousness, which would require it to be self-aware. Because we are not yet able to adequately define what consciousness is, it is unlikely that we will be able to create a conscious AI soon." (IBM, 2019)

2.1.3 The role of artificial intelligence in software engineering

AI is revolutionary in improving software quality, accelerating productivity, and increasing project success rates. AI can assist software teams in many ways, e.g. automating routine tasks, providing project analytics and actionable recommendations, and even making decisions. Software has increased in both size and complexity, which emphasizes the need of AI tools. It is important to note that AI is not trying completely to replace human teams, it will be more like as assistance to human and tool to warrant trust. (Dam, 2019)

Most businesses are showing even more interest to AI. There are high growth forecasts:

- 80% of companies are investing to AI and have some form of AI (Teradata, 2017)
- AI-enabled tools will generate $2.9 trillion in business value and 6.2 billion hours of worker productivity globally by 2021 (Gartner, 2019a).
- In 2019, organizations working with AI have on average 4 AI projects in place. It is forecasted that in 2021 the number of projects is 20 and in 2022 it is 35. (Gartner, 2019b)
- Figure 5 presents how worldwide revenues for the AI market are expected to grow enormously over the next ten years (Statista, 2020).

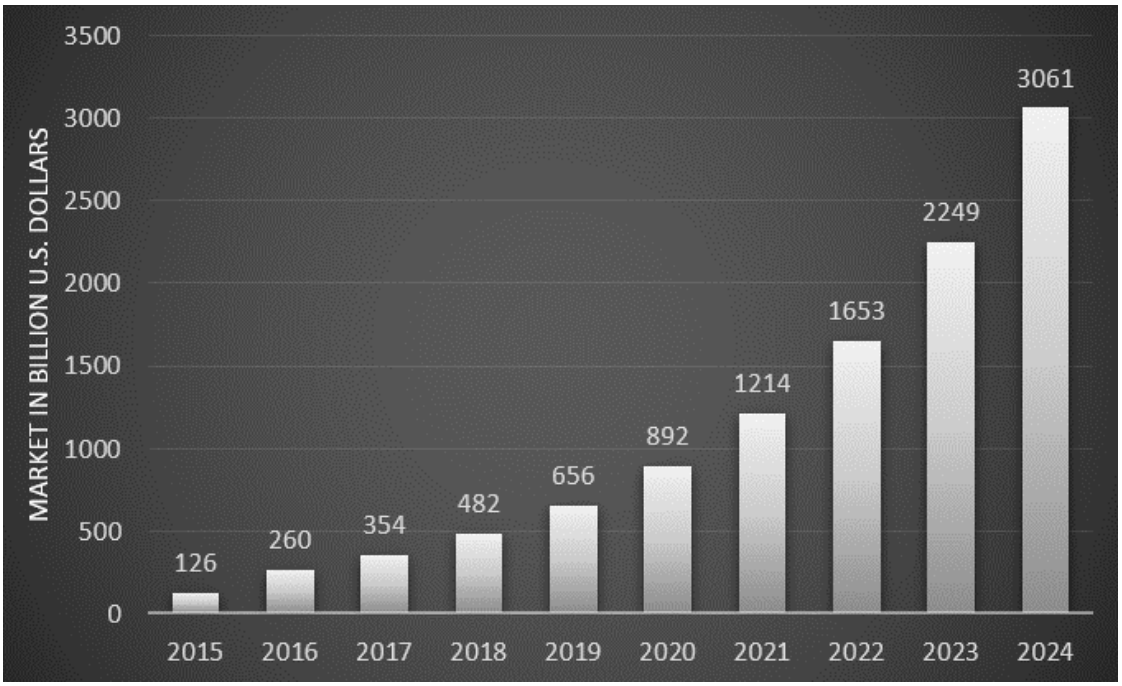

Figure 5. Worldwide revenues for the AI market from 2015 to 2024 (adapted from Statista, 2019)

Nevertheless, there are still big obstacles to adoption. Survey conducted in 2018 reports that only a small proportion of respondents recognize that their organization has enough trained people internally to buy, build and deploy AI. Top barriers to AI were a lack of IT infrastructure, a lack of access to talent and understanding AI use cases. (Gartner, 2019a)

2.1.4 Programming languages for artificial intelligence

Developers have a variety of programming languages to use in coding AI. There is no single best programming language, it is up to developer to choose suitable to match application requirements. According to Existek (2018), the top five major AI programming languages are:

- **Python**

  Python syntax is simple and versatile, which makes it quite fast in development. It is portable and can be used on many platforms. There is extensive variety of libraries and tools. Object-oriented design increases a programmer's productivity. The downside is that Python is not suitable for mobile computing. Python works with the help of an interpreter which makes compilation and execution slower in AI development. (Existek, 2018)

- **C++**

  C++ is the fastest computer language, so it fits well for projects that are time sensitive. It supports reuse of programs in development due to inheritance and data hiding. It is appropriate for machine learning, neural networks, and complex AI problems. The major drawback is that C++ is highly complex, making it hard for newcomer developers. In addition, C++ is not so good in multitasking. Therefore, it is suitable for only implementing the core or the base. (Existek, 2018)

- **Java**

  Java is generally one of the most used programming languages and it is very portable. It is simple to use and debug. Automatic memory manager eases the work of the developer. Java is not appropriate for neuro-linguistic programming, search algorithms, and neural networks. It has less speed in execution and thus more response time than C++. It has the disadvantage that older platforms would require changes on software and hardware to facilitate. (Existek, 2018)

- **Lisp**

  Lisp is the second oldest programming language. It is fast and efficient in coding as it is supported by compilers instead of interpreters. It is used in AI because of its flexibility for fast prototyping and experimentation. It is appropriate to use for inductive logic projects and machine learning. The drawback is that few developers are very familiar with Lisp programming. Additionally, Lisp requires configuration of new software and hardware to accommodate its use. (Existek, 2018)

- **Prolog**

  According to Existek, Prolog is one of the oldest programming languages. Its strengths are that it is fast for prototyping and allows database creation simultaneously with running of the program. The disadvantage is that it has not been fully standardized. In consequence, some features differ in implementation, making the work of the developer laborious. (Existek, 2018)

2.2 Machine learning

Machine learning (ML) has seen as a subset of AI and it is one of the most important branches of AI (Ertel, 2011). Machine learning is one area of AI that has experienced major breakthroughs in recent years, mostly due to the growth of big data and increased computational power (Dam, 2019). In 1997, Tom Mitchell defined machine learning as follows (Ertel, 2011):

"Machine learning is the study of computer algorithms that improve automatically through experience."

Machine learning builds models to classify and make predictions from data. Training refers to using a learning algorithm to determine and develop the parameters of your model (IBM, 2019). It is important to split data to a training dataset (experience) and a testing dataset. The training dataset is used to train the model and it contains knowledge which the learning algorithm is supposed to extract and learn. The testing dataset is unknown data, and it tests the generalization ability of the learning algorithm. Otherwise every system would perform optimally just by calling up the saved data. The testing dataset is used to evaluate how good

our model is, using terms like, accuracy and precision. Machine learning relies on defining behavioral rules by examining and comparing large datasets to find common patterns. The model will improve in performance as it gathers more experience, in other words, it gets more data. (Ertel, 2011)

Machine learning is applied in a variety of fields: robotics, natural language processing, product recommendation, e-mail spam filtering, medical diagnosis, computer games, and many others. (IBM, 2019) Figure 6 shows the three main types of learning problems in machine learning: supervised learning, unsupervised learning, and reinforcement learning.

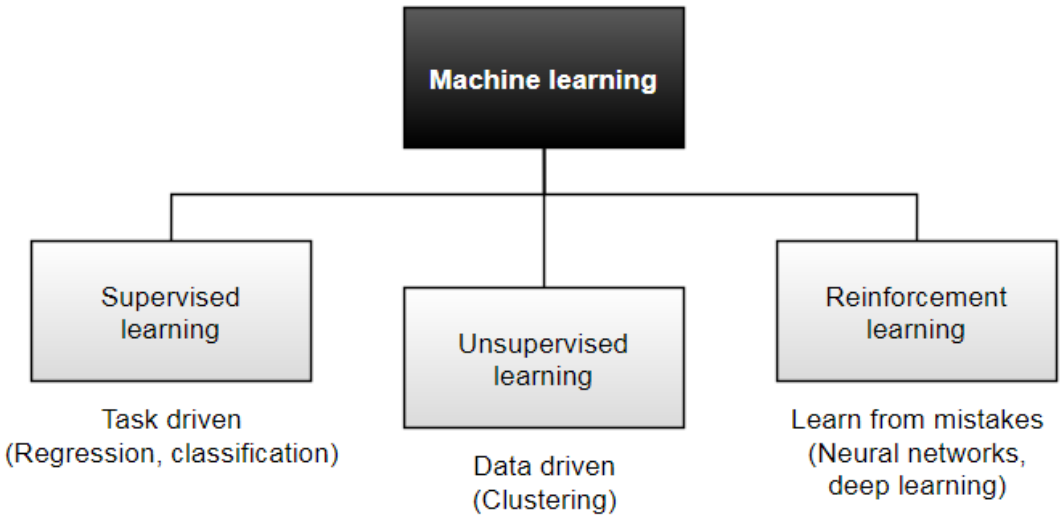

Figure 6. Types of machine learning

2.2.1 Supervised learning

In supervised learning, we give input and the model task is to predict the correct output or label. Only inputs are provided and outputs from the model are compared to known correct labels and used to estimate the skill of the model. In the optimal scenario, the model can correctly predict a class label for unseen instances. It is called supervised learning because of the idea of a teacher supervising the learning process. The teacher knows the correct answers and the model iteratively makes predictions on training data and is corrected by the teacher. (Brownlee, 2019) In its most basic form, a supervised learning algorithm can be written as:

$$y = f(X)$$

Where *y* is the predicted output that is determined by the function that takes the input value *X*. The function is created by machine learning algorithm during training. (Wilson, 2019)

There are two main types of supervised learning: regression and classification.

- Regression is a supervised learning problem that invokes predicting numerical values. An example of a regression problem would be predicting house prices. Inputs are variables that describe the house information and output is the house price. (Brownlee, 2019) The three most common types of regression algorithms are linear regression, logistic regression, and polynomial regression. Figure 7 (a) shows the result of a linear regression algorithm. You can notice that there is a linear correlation between x_1 and x_2, and the line of best fit can be drawn through the data points. You can use the line of best fit to predict output values. (Wilson, 2019)

- Classification is a supervised learning problem that involves predicting a class label. An example of a classification problem would be predicting handwritten digits. There inputs are images of handwritten digits (pixel data) and the output is a class label for what the image represents, numbers between 0 to 9. (Brownlee, 2019) For example, a few popular classification algorithms are linear classifiers, support vector machines, decision trees, k-nearest neighbors, and random forests. (Wilson, 2019) Figure 7 (b) shows the result of a linear classifier classification algorithm where the line separates two classes from each other. You can use the line to predict which class new input belongs to.

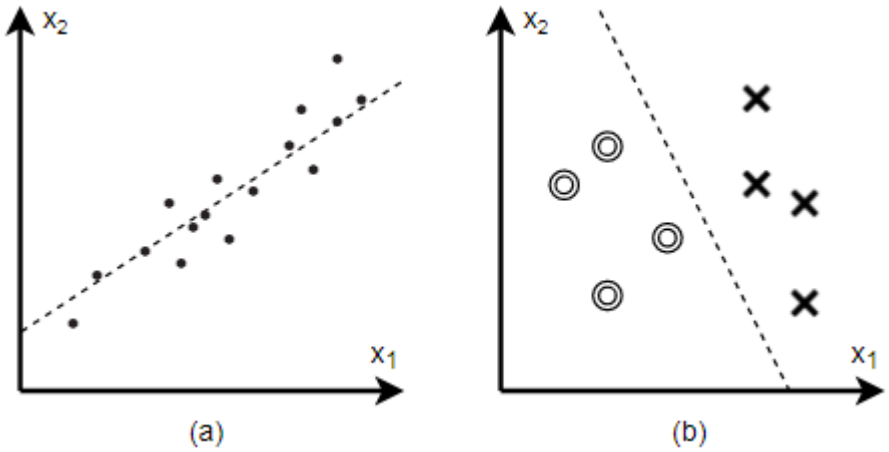

Figure 7. Supervised learning: a) regression b) classification

2.2.2 Unsupervised learning

According to Brownlee (2019), unsupervised learning is a machine learning technique, where you do not need to supervise the model. There is no teacher because data are unlabeled. Unsupervised learning is used to draw inferences and patterns from datasets by itself. The advantage of unsupervised learning is that you do not need to label data, which includes manual work. Unsupervised learning is appropriate to use when you do not know how many classes data are divided into, unlike supervised learning. (Brownlee, 2019)

The most popular unsupervised learning method is cluster analysis. It tries to find hidden patterns and group data. As stated in MathWorks (n.d.), a clustering algorithm can discover groups of objects where the average distances between the members of each cluster are closer than members in other clusters. Figure 8 presents the result of a clustering algorithm, where it found two clusters. Referring to MathWorks (n.d.), the most common clustering algorithms are hierarchical clustering, k-means clustering, gaussian mixture models, self-organizing maps, and hidden Markov models.

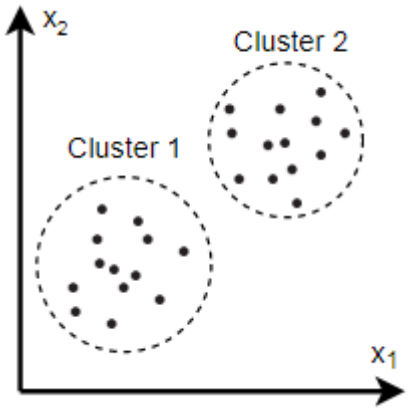

Figure 8. Unsupervised learning: clustering

2.2.3 Reinforcement learning

Reinforcement learning is a type of machine learning where a computer learns to achieve the goal through repeated trial-and-error interactions with a dynamic environment. You need to define the state, the desired goal, allowed actions, and constraints. (IBM, 2019) Dickson (2017) stated that a reinforcement learning algorithm figures out how to achieve the goal by trying different combinations of allowed actions. It is rewarded or punished depending on whether the decision was a good one. The algorithm tries its best to maximize its rewards within the constraints provided. For example, you could use reinforcement learning to teach a machine to play games, like chess. (IBM, 2019)

Figure 9 presents the flow of reinforcement learning. For example, let us take and consider dog training. In this case, the dog is the agent and the surroundings of the dog represent the environment. First, the trainer gives a command that the dog observes. Then the dog responds by taking an action. If the action is a desired behavior, the trainer will provide a reward. (Tzorakoleftherakis, 2019)

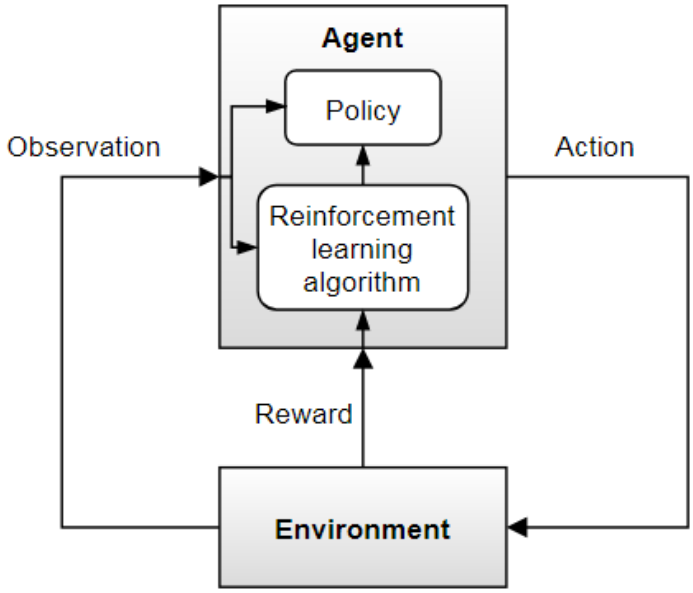

Figure 9. Agent-environment interaction in reinforcement learning (Tzorakoleftherakis, 2019)

2.3 Deep learning

According to Dickson (2017), deep learning is a specialized subset of machine learning. Deep learning layers algorithms to create a neural network, an artificial replication of the structure of the brains, enabling AI systems to continuously learn on the job and improve the quality and accuracy of the results. Deep learning can learn from unstructured data such as photos, videos, and audio files, and it has proven to be very effective at various tasks. (IBM, 2019) The efficiency and performance of earlier learning algorithms tend to remain the same when more training data is used, in contrast to deep learning. IBM (2019) stated that deep learning algorithms continue to improve as they are fed more data.

Advances in machine learning achieved within the recent years by combining massive data sets and deep learning techniques (Elements of AI, 2019). Neural networks are the reason why deep learning algorithms can continuously learn on the job and improve the quality and accuracy of results as datasets increase in volume over time. (IBM, 2019) Neural networks are networks of nerve cells in the brains of human. For centuries, researchers tried to understand how the brain functions. The first big step was made in 1943 by McCulloch and Pits. They presented a mathematical model of the neuron as the basic switching element of

the brain. Their publication was the foundation for the construction for artificial neural networks. (Ertel, 2011)

2.3.1 Artificial neural network

An artificial neural network (ANN) is a collection of smaller units called neurons. As reported by IBM (2019), artificial neural networks borrow some ideas from the biological neural network of the brain. These neurons take input data and learn to make decisions over time. Artificial neural networks learn through a process called backpropagation. Backpropagation uses a set of training data that is labeled, so it can match known input to desired output. (IBM, 2019)

IBM (2019) simply presents the backpropagation process as follows:

1. First, inputs are plugged into the network, and outputs are calculated.
2. Secondly, an error function determines how far the given output is from the desired output.
3. Finally, adjustments are made to the weights and the bias to reduce error.

Figure 10 represents a deep neural network. It is called “deep” because there is two or more hidden layers. Neurons are represented by the nodes and an arrow shows the relationship between the nodes. A collection of neurons is called a layer and every neural network will have one input layer and one output layer. Furthermore, it will have one or more hidden layers. An input layer forwards input values to the next layer. Hidden layers take weighted input and produce output through an activation function. Deep learning networks end in an output layer to predict a particular outcome or label. For example, the input data are 90 percent likely to represent a dog. (IBM, 2019) The disadvantage of ANNs are that they cannot capture sequential information from input data. Moreover, there is a vanishing and exploding gradient problem when performing backpropagation. (Pai, 2020)

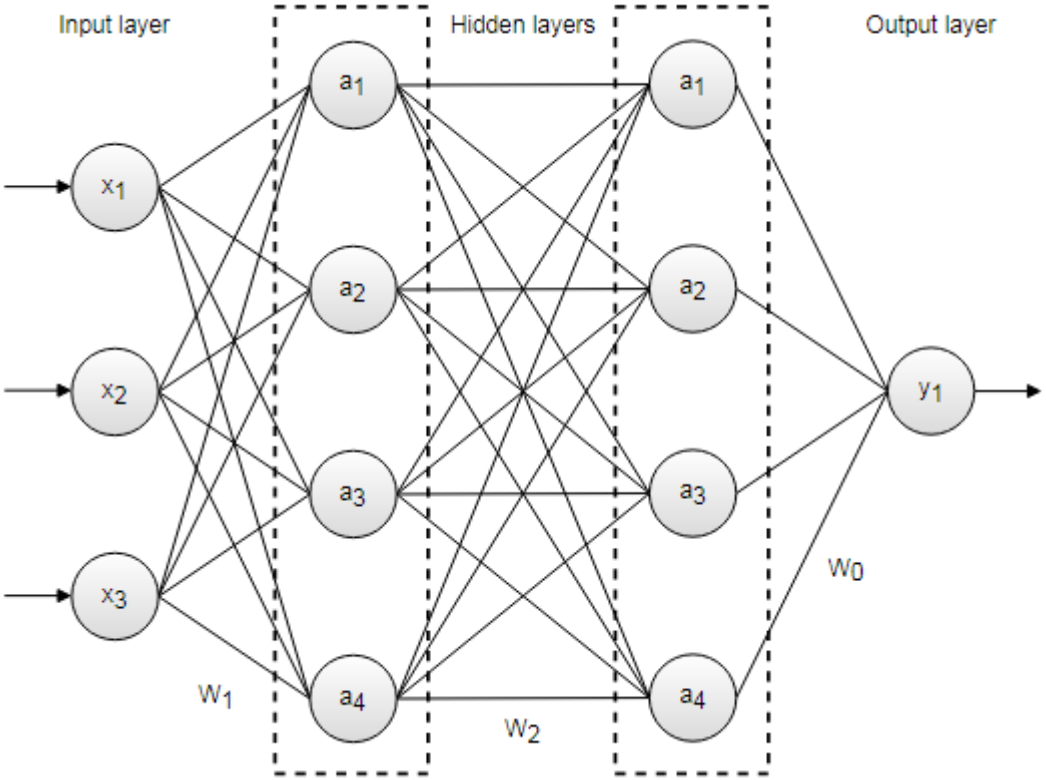

Figure 10. Deep neural network (adapted from IBM, 2019)

2.3.2 Convolutional neural network

According to IBM (2019), a convolutional neural network (CNN) is one of the variants of neural networks used heavily in applications such as image processing, video recognition, and natural language processing. IBM (2019) describes a convolution as follows:

"*A convolution is a mathematical operation, where a function is applied to another function and the result is a mixture of the two functions. Convolutions are good at detecting simple structures in an image and putting those simple features together to construct more complex features.*"

The advantage of the CNN is that it can recognize an object anywhere in the data no matter where it has been observed in the training data. You no longer need so big training data to gain high accuracy because you do not need to teach every possible place for the object individually. For example, cat's ears can appear in different positions, different orientations, and in different sizes in the image. (Elements of AI, 2019)

2.3.3 Recurrent neural network

According to Nabi (2019), recurrent neural networks (RNNs) are called recurrent because they perform a same task for a every element of the sequence, with output being depended on the previous computations. RNNs are used for speech recognition, voice recognition, time series prediction, and natural language processing. For example, you can use an RNN to predict what is the next word in a sentence. Normally, ANNs consider only the current input and cannot handle sequential data, unlike RNNs. RNNs can memorize previous inputs due to their internal memory. Figure 11 presents a simple recurrent neural network, where the *x* is the input layer, the *h* is the hidden layers and the *y* is the output layer. (Biswal, 2020)

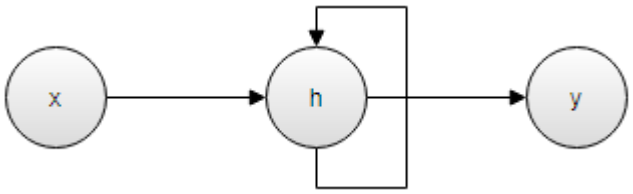

Figure 11. Recurrent neural network (adapted from Biswal, 2020)

Long short-term memory (LSTM) is a breakthrough of an RNN. LSTMs are a special kind of the RNN, which are capable of learning long-term dependencies. LSTMs are smart in remembering things that have happened in the past and finding patterns across time to make its next guess. (Pai, 2020)

2.3.4 Summary and comparison

It can be difficult for a beginner to select a suitable neural network. Pai (2018) and Brownlee (2018) gave instructions to facilitate this problem, shown in Table 2. It presents, among other things, which neural network is applicable for certain type of data.

Table 2. Summary of neural networks (adapted from Pai, 2018; Brownlee, 2018)

	ANN	**CNN**	**RNN**
Use for	Tabular datasets, classification, and regression prediction problems.	Image data, classification, and regression prediction problems.	Text data, speech data, classification and regression prediction problems, generative models.
Try on	Images, texts, time series data, and other types of data.	Text data, time series data, sequence input data.	Time series data
Do not use	-	-	Tabular data, image data
Recurrent connections	No	No	Yes
Spatial relationship	No	Yes	No
Vanishing & exploding gradient	Yes	Yes	Yes

2.4 Refactoring

Requirements for a software system in use evolve which is caused by changes in the business model and with the demands of users in terms of software functionality. Software is so complex today that you cannot develop a system architecture that will meet all requirements from the start. Because of this, you need to take care of software reusability, maintainability, and extensibility. One technique for that is refactoring of existing software. This technique consists of multiple transformations that you can use to improve system architecture significantly and consistently. This improvement is then the basis for extending the system. (Rumpe, 2017) The term "refactoring" can be used either as a noun or a verb. Refactoring is often defined as follows:

- Refactoring (noun): "*A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*." (Fowler, 2018)

- Refactoring (verb): "*To restructure software by applying a series of refactorings without changing its observable behavior."* (Fowler, 2018)

2.4.1 Reasons for refactoring

Fowler presented (2018) several purposes why developers should refactor:

- **Refactoring improves the design of software.** Without refactoring, the design of software will decay. One way to refactor is to remove duplicate code. It will not make the system run much faster but make a big difference in modification of the code. The more code there is, the harder it is to modify. (Fowler, 2018)

- **Refactoring makes software easier to understand and more readable.** Developers often do not think about a future developer who will need to edit or use that code. In the worst case, it would take a week the developer to make a change, what would have taken an hour if the new developer had understood the code. (Fowler, 2018)

- **Refactoring helps a developer to detect bugs.** When you refactor, you deeply understand the code. Understanding also helps finding bugs. Refactoring can help developers to be much more effective and write well-made code. (Fowler, 2018)

- **Refactoring helps a developer to program faster.** Improving design, improving readability, reducing bugs lead the developer to code fast. It is essential especially for rapid software development. It stops the design of system from decaying because developers can develop software more rapidly. If the code is clear, the developer is less likely to introduce a bug, and if the developer does, the debugging is much easier. (Fowler, 2018)

Regardless, you do not need to always refactor, sometimes it is valuable. You should only refactor when refactoring gives you any benefits. In some cases, it is easier to just rewrite than refactor. Decision whether to rewrite or refactor requires good judgement and experience. (Fowler, 2018)

2.5 Code smells

According to Fowler (2018), deciding when to start refactoring, and when to stop it is just as important factor in refactoring as knowing how to operate it. Kent Beck and Martin Fowler came up the idea of describing “when” of refactoring in terms of different smells. Fowler (2018) defines a code smell as follows: “a code smell is a surface indication that usually corresponds to a deeper problem in the system”. Code smells do not affect output, but they make code hard to maintain and adapt to new requirements. Code smells are removed from code using refactor techniques. Bad program design, implementation choices, and bad programming practices are the common causes of code smells. (Tufano et. al., 2017)

2.5.1 Types of code smells

Table 3 presents an overview of code smells introduced by Fowler (2018). Some of the code smells were explained in more detail on the Refactor.Guru (n.d.) website than in the Fowler's book. Consequently, I used the Refactor.Guru (n.d.) to support code smell descriptions.

Table 3. Code smells (adapted from Fowler, 2018; Refactoring Guru, n.d.)

Code smell	Description
Mysterious name	Code needs to be mundane and clear. One of the most important parts of code is good names. Developers need to put plenty of thought into naming functions, modules, variables, and classes so that they clearly communicate what they do and how to use them. The most common refactoring is to rename a function. (Fowler, 2018)
Duplicated code	Duplicated code means that you notice the same code structure in more than one time. A program will be better if you can unify them. Duplicated code causes extra work when you need to change the duplicated code, because you must find and catch each duplication. (Fowler, 2018)
Long function	People have realized that the longer a function is, the more difficult it is to understand. Developers should be more aggressive about decomposing functions. You can identify a long function by a comment that tells what it is doing. It can often be replaced by a method based on the comment. (Fowler, 2018)
Global data	Developers are always warned using global data. The problem of global data is that it can be modified from anywhere in the code base, and there is no mechanism to discover which bit of code touched it. They often lead to bugs that are hard to find out where they originate. It gets exponentially harder to deal with the more you have. (Fowler, 2018)

Continuation of Table 3

Code smell	Description
Mutable data	Mutable data can be changed after it is created. Problems occur when you update some data and do not realize that another part of the software expects something different and now fails. Mutable data is not a big problem when it is a variable whose scope is just a couple of lines, but its risk increases as its scope grows. (Fowler, 2018)
Divergent change	Divergent change occurs when you need to do too many changes in a class/module to introduce a new feature or change. Usually you want to do change only in one point. Common way to refactor is to extract a class or function. (Fowler, 2018)
Shotgun surgery	"Shotgun Surgery refers to when a single change is made to multiple classes simultaneously. Making any modifications requires that you make many small changes to many different classes." (Refactor.Guru, n.d.)
Feature envy	"A classic case of feature envy occurs when a function in one module spends more time communicating with functions or data inside another module than it does within its own module." (Fowler, 2018)
Long parameter list	Long parameter lists are often confusing. You can obtain one parameter by asking another parameter for it, so you can remove the second parameter. Classes are a great way to reduce parameter list sizes. Especially when multiple functions share several parameter values. (Fowler, 2018)
Primitive obsession	Primitive obsession is when the code relies too much on primitives, like integers, floating point numbers, and strings. You should not use primitives instead of small objects, such as money, coordinates, or ranges. (Fowler, 2018)

Continuation of Table 3

Code smell	Description
Repeated switches	The same conditional switching logic pops up different places. The problem with such duplicate switches is that, whenever you add a clause, you must find all the switches and update them. (Refactoring.Guru, n.d.)
Loops	Loops are no more relevant these days because first class functions are widely supported. Pipeline operations such as filter and map can help quickly to see the elements that are included. (Fowler, 2018)
Lazy element	For example, a class that is one simple function or class that after some of refactoring has become small. If it does not do enough, it should be deleted. (Fowler, 2018)
Speculative generality	Occurs when there is unused class, method, field, or parameter. (Refactoring.Guru, n.d.)
Temporary field	For example, a class in which field is set only in certain circumstances. Outside of these circumstances, they are empty. It makes code hard to understand, you expect to see data in object fields. (Fowler, 2018)
Message chains	Occurs when you see message chains when a client asks one object for another object, which the client then asks for yet another object, and so on. (Refactoring.Guru, n.d.)
Middle man	Class that delegates work to other classes. (Fowler, 2018)
Insider trading	Modules that whisper to each other. Trading data around too much increases coupling. (Fowler, 2018)
Large class	Class that is doing too much. Too many fields/methods/lines of code. In addition, it helps duplicated code emerge in class. (Fowler, 2018)
Alternative classes with different interfaces	"Two classes perform identical functions but have different method names. The programmer who created one of the classes probably did not know that a functionally equivalent class already existed." (Refactoring.Guru, n.d.)

Continuation of Table 3

Code smell	Description
Data class	Classes that have fields, getting and setting methods for fields, and nothing else. They are just holding data. (Fowler, 2018)
Refused bequest	"If a subclass uses only some of the methods and properties inherited from its parents, the hierarchy is bizarre. The unneeded methods may simply go unused or be redefined and give off exceptions." (Refactoring.Guru, n.d.)
Comments	Comments are often used as a deodorant to hide code smells. Additionally, comments are often there because the code is bad. However, that does not mean all the comments are bad. (Fowler, 2018)
Data clumps	Different parts of code contain same variables. Bunches of data that likes to hang around together. (Fowler, 2018)

2.5.2 Negative effects and concerns

Sae-Lim et al. (2017) presented negative effects of code smells in their research:

- Classes with code smells are more likely to change and become faulty.
- Code smells significantly decreased the performance of developers.
- The relationships between code smells were related to maintenance problems.

There have been multiple surveys on how developers are concerning code smells (Sae-Lim et al., 2017):

- Developers said that improved tools to detect code smells, especially tools with context-sensitive features, are needed.
- The fear of breaking the client code was one explanation why code smells remain in source code.

- Developers perceived code smells differently as problems according to the types of code smells.
- Developers were aware of code smells, but they were unlikely to solve them.

2.5.3 Detection tools

Nowadays, there are an increasing number of software analysis tools for detecting code smells. The automatic tools play relevant role in finding code smells in large code bases. The tools are important because many code smells can go unnoticed while programmers are working. (Fontana et al., 2012)

The concept of a code smell is vague and prone to subjective interpretation. For example, a word “large” is ambiguous and hard to define exactly. Different tools use different techniques and therefore give different results. Furthermore, tools use different threshold values for the metrics to detect code smells. A reliable tool should return precise and reproducible answers and prioritize them. (Fontana et al., 2012) There is a problem with accuracy of the result, many false positive smells can be detected because of the information related to the whole system is not considered (Fontana et al., 2016).

Almost all detectors identify code smells using structural properties of source code and extracting feature set from the source code. Shortcoming for using these code metrics is that we need an external tool to calculate the code metrics for the specific programming language. Source code metrics can be, for example, the cyclomatic complexity and the number lines of code. For that reason, applying a ML method is redundant because a tool can deduce smells directly by combining these metrics. Secondly, using only these metrics is limiting a ML algorithm, because it cannot observe any patterns that is not captured by an external metric tool. (Sharma et al., 2019)

Past research on code smell detection can be divided into two main categories, the rule-based approaches, and the ML approaches. Both approaches are useful and equally good ones. Researchers believe it is unlikely that a single technique can completely solve the code smell problem. For example, when comparing to email spam filtering, there are over 30 different techniques to detect email spam. (Fontana et al., 2016)

1. **Rule-based approaches**

   The rule-based approaches rely mostly on the metrics. It requires that engineers create specific rules for defining each smell. Rule creation task requires effort from the engineers. There is often misalignment between what is considered smelly by the tool and what is refactorable by the engineers. (Fontana et al., 2016)

2. **ML approaches**

   ML technology can be used to make tool learn how to classify code smells by examples. In addition, the ML approaches rely mostly on the metrics. In the ML-based approaches, the ML algorithms create rules and engineers only provide the information whether a piece of code has a smell or not. The ML approaches can be used to make less subjective definition of code smells. The application of ML to the detection of these code smells can provide high accuracy, and only a hundred training examples were needed to reach at least 95% accuracy. (Fontana et al., 2016) Researchers have used Bayesian belief networks, deep learning models, support vector machines, and binary logistic regression to identify code smells. The ratio of negative and positive samples affects how well a ML based approach can perform. When the ratio is balanced, classification becomes easier. In the real world, the ratio can be up to 182 to one. (Sharma et al., 2019)

2.6 Natural language processing

Unstructured data have no specific format and no data model. Text data, image data and video data are some of the examples of the unstructured data. These types of data are estimated to represent 80 percent of the valuable information for most of the organizations. (Gupta, 2019) In source code, most of the data are unstructured data, such as natural language text in comments and identifier names. According to Gupta (2019), researchers in the software engineering community have developed many techniques for handling such unstructured data, such as natural language processing (NLP). Spellcheck and autocomplete are common applications of NLP. Carnes (2020) defines NLP as follows:

"Natural language processing is a discipline in computing that deals with the communication between natural (human) languages and computer languages."

Open source projects are more popular than ever. There is huge amount of public data to analyze with open source tools, like GitHub. These source codes and meta-data, such as changes, bug-fixes, and code reviews are also called as the "big code". This offers a great resource of software engineering data for researchers. Most of the NLP research in software engineering is focused on software process documents, archived communications, discussions in question-answering sites, source code and mobile app store reviews. (Gupta, 2019)

2.6.1 Differences between text and code

There are major differences that you need to be taken into consideration when you are using source code in your research. Studies that apply deep learning on the source code rely heavily on results from the text mining domain. (Sharma et al., 2019) The source code has two audiences. It communicates with humans and with computers. Humans must understand the code, and computers to execute it. Allamanis et al. (2018) presented text and code differences to get an idea when NLP techniques need to be modified to handle the code.

- **Executability**. Code is executable, text often is not. The code is semantically fragile, if you change one small bit of the code, it can change dramatically the meaning of

the code. As a natural language is more robust, a reader can understand it even if it contains mistakes. Executability allows you to track execution traces, which are not present in the text. (Allamanis et al., 2018)

- **Formality.** Programming languages are formal languages, whereas formal languages are only mathematical models of a natural language. Code is more pattern dense than text and it must be unambiguous. Natural languages change gradually while a programming language changes abruptly in new releases, e.g. Python 3. The formality of the source code eases the reuse of the code. (Allamanis et al., 2018)

- **Cross-channel interaction.** Code has two channels, the algorithmic and the explanatory. Identifiers, statements, blocks, and functions cannot be mapped universally to textual semantic units. A function differs from a paragraph, in that it is named and called. Paragraphs rarely have names or are referred to elsewhere in text. Parse trees of paragraphs in text tend to be diverse, short, and shallow compared to abstract syntax trees of functions. (Allamanis et al., 2018)

2.6.2 Representing code

Allamanis et al. (2018) showed that we have many ways to represent code:

- **Token-level models.** Token-level models view code as a sequence of tokens or characters. They are commonly used because of their simplicity. (Allamanis et al., 2018)

- **Syntactic models.** Syntactic models represent code at the level of abstract syntax trees (ASTs). ASTs tend to be deeper and wider than text parse trees due to the highly compositional nature of the code. (Allamanis et al., 2018) Figure 12 illustrates how the code sample can be transformed to the AST.

- **Semantic models.** Semantic models view code as a graph. Graphs are natural representations of source code and require little abstraction. Generating a complex

graph is hard because there is no starting point like trees have. (Allamanis et al., 2018)

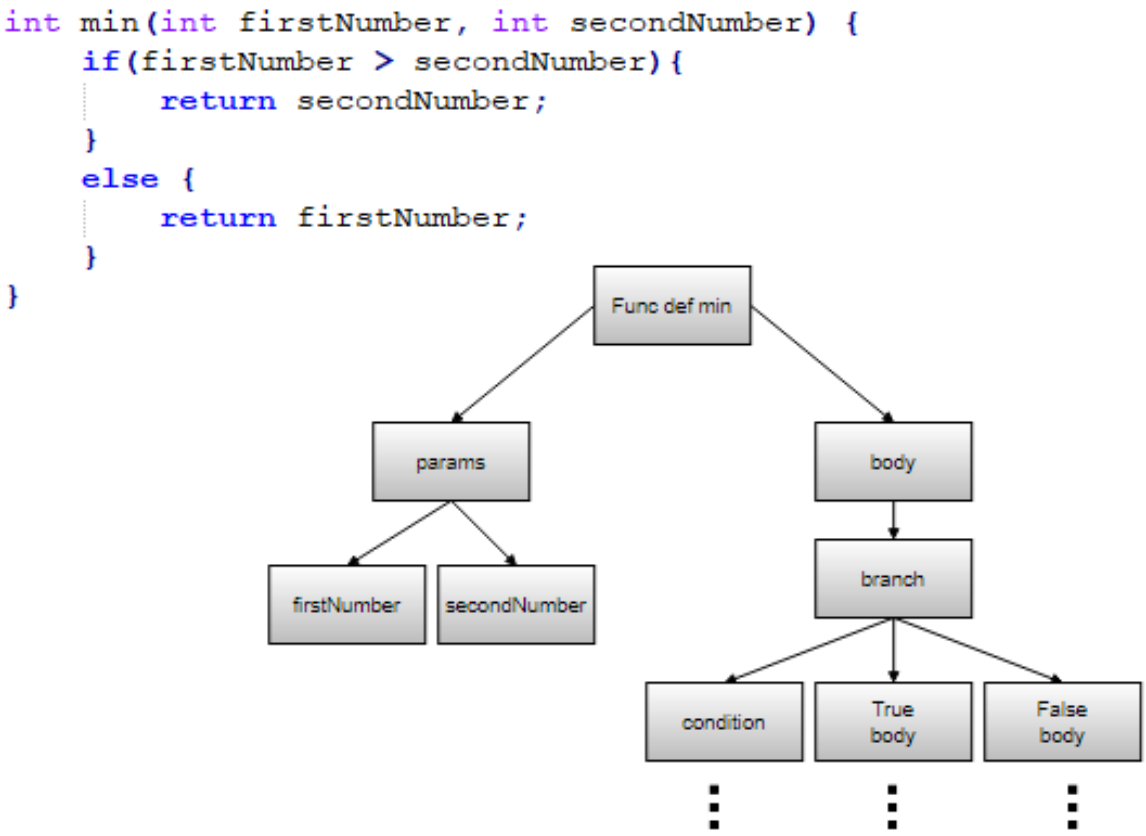

Figure 12. Transformation of the code sample to the AST (adapted from Ďuračík et al., 2017)

2.6.3 Encoding text

As machines cannot understand text data in a raw form, input of a ML algorithm needs to be numerical. There are multiple techniques to encode text and its features as numbers:

- **Encode each word (or character) to a unique number**. In this approach, we create a vocabulary for each unique word in a sentence. For example, the word "cat" represents number 1 and the word "mat" represents number 2, and so on. Finally, we can transform sentences to a dense vector, like [1, 5, 2, 1], where all the values are non-zero. The downside is that it does not capture any relationship between words, and it can be challenging for a model to interpret. (TensorFlow, 2019) Figure 13

represents how an open-source tool Tokenizer converts source code into vectors of numbers.

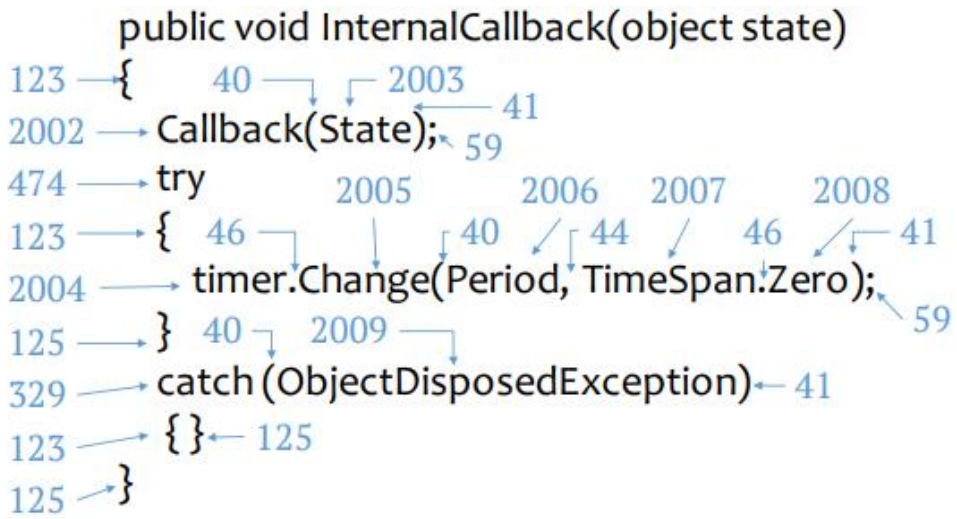

Figure 13. Tokens generated by Tokenizer (Sharma et al., 2019)

- **One-hot encoding**. This approach uses the same unique word vocabulary. To represent each word, we create a vector length equal to the vocabulary, then place a one in the index that corresponds to the word. As a result, we get many sparse vectors, where most indices are zero, like [0, 0, 1, 0, 0]. If we have 10 000 words vocabulary, each word would be a vector where 99,99% of the elements are zero. Because of that, this approach is inefficient. (TensorFlow, 2019)

- **Bag of words**. This approach keeps tracks of words frequency. It uses the same unique word vocabulary. Every time a word appears in a sentence, its count is increased by one. The downsides are that we lose information on the grammar and ordering of the words in text. Furthermore, the same as one-hot encoding, the length of the vectors can grow big when the vocabulary grows and there are many zeros. For example, a sentence could be transformed to [2, 0, 4, 1, 0], where number represents the frequency of the indexed word. (Huilgol, 2020)

- **TF-IDF (Term Frequency-Inverse Document Frequency)**. TF-IDF is a scoring scheme for words. It measures how important a certain word is to a document. For

example, “the” word is often in every sentence and thus its significance for the sentence is unimportant, and it would get low weight. (Huilgol, 2020)

- **Word embedding**. Word embedding keeps the order of words intact as well it encodes similar words with very similar labels. Each word is represented by vector. Words that are used in similar ways result in having similar representations, capturing their meaning and relation with other words. You can add an embedding layer to the beginning of the model, and while training the embedding layer will learn correct embeddings for words. There are also pretrained embedding layers available to use. (TensorFlow, 2019)

3 PROTOTYPE DESIGN

This chapter explains my prototype design. My intended goals, a vision, and requirements are described. For example, I clarify why I selected these certain decisions and what different alternative approaches I considered. Typically, projects start with planning and gathering requirements.

3.1 Vision of the solution

My vision from the beginning was to use deep learning. I did not want to use any external tools to calculate metrics from source code or make rules to detect code smells. I wanted to keep it as natural as possible and therefore handle code as characters or tokens. Figure 14 represents the overview of the design.

It is easy to find open source code repositories from GitHub and get code samples. However, it requires work and some experience to detect code smells. I was hoping that there were already comprehensive code smell datasets, or I would be able to use existing tools to detect code smells.

Tokenizing data is required to transform text to numerical form so that a ML algorithm can understand and use it. There should be many tokenizers available from the NLP field and even coding own tokenizer should not be so hard.

There are many neural networks to choose from, like ANNs, RNNs and CNNs. Code and text are sequential data. RNNs should work best with sequential data because there are connections between nodes, and they can use their internal memory to process sequences of inputs. In RNNs all inputs are related to each other, in contrast to other neural networks, where inputs are independent of each other. A special kind of RNN, LSTM is often used because it is capable of learning long-term dependencies. It can remember past data in memory when the gap between relevant information becomes large. This can happen often with source code tokens.

Finally, when the model is trained with some data, it should be able to make predictions about any source code samples and tell whether they smell, or they do not smell. The sigmoid activation function gives the output value between 0 and 1. Therefore, the output can be displayed as a percentage, e.g. “your code smells with a 96% probability”.

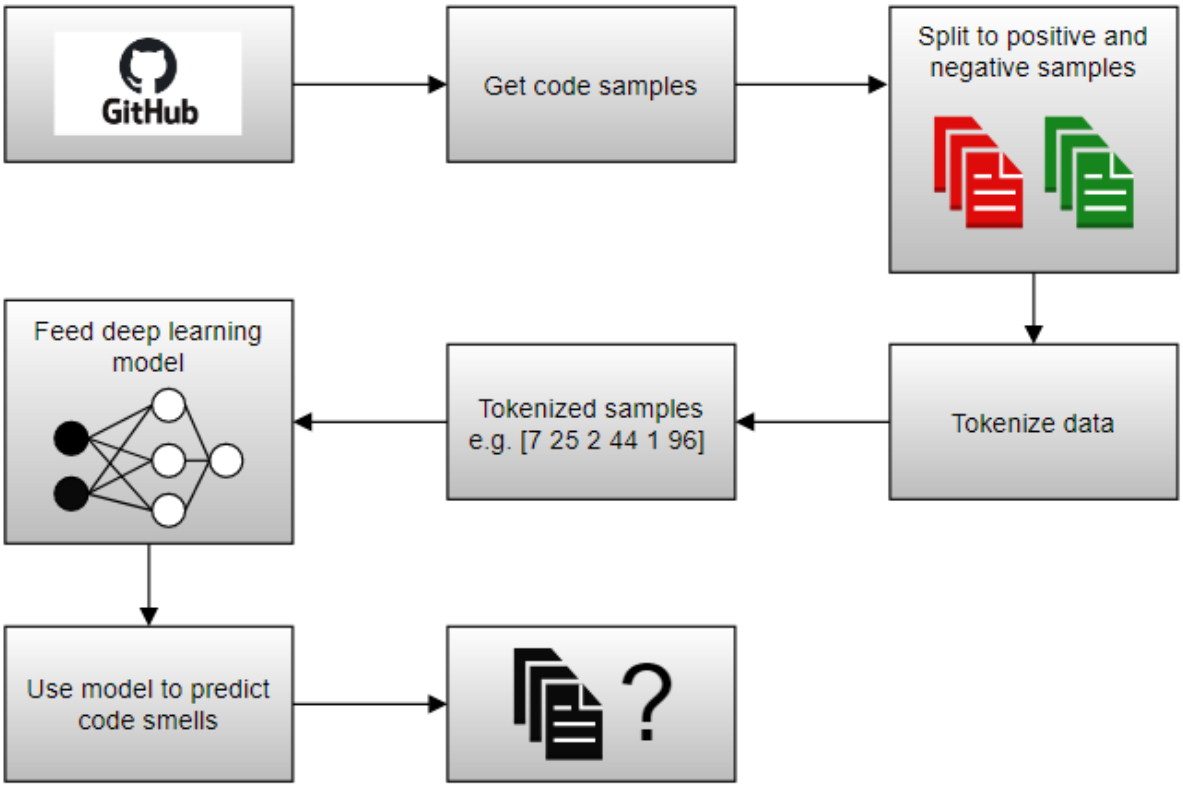

Figure 14. Overview of the design

3.1.1 Vision statement

The vision statement describes the core and overall objective of my solution. It explains why my prototype exists and where it is going.

“It is aimed for developers who need to find code smells in their source code and want to refactor it to achieve more reusable, maintainable and extensible software. This prototype is a source code analyzer. It can tell whether your code smells or not. Unlike the most code smell detectors, this prototype is neither rule-based nor metric-based. This prototype uses tokenized and labeled data to train the deep learning model.”

3.1.2 Major features

The major features of the prototype are listed below:

1. Load code smell dataset from a CSV (comma-separated values) file.
2. Tokenize data.
3. Create a model.
4. Train a model.
5. Make prediction whether a method-level code sample smell or not.
6. Save and load a model.

3.1.3 Assumptions and dependencies

These are the assumptions and the dependencies I made before the development of the prototype:

- Assumption: Deep learning is feasible to use.
    - Dependency: The prototype accuracy is not certainly good.

- Assumption: 20 code smells and 20 non-smelly code samples are big enough dataset to train a model.
    - Dependency: The accuracy of the model may not be good as it could be with more data.

- Assumption: High severity code smells are easier to detect than lower severity code smells.
    - Dependency: Dataset will consist of crucial severity code smells to achieve higher accuracy.

- Assumption: Graphical user interface (GUI) is not needed to this initial version.
    - Dependency: The program is executable, and text based.

- Assumption: There might be differences in detecting different code smells.
    - Dependency: The accuracy of certain code smells can change.

- Assumption: RNNs are better than CNNs or ANNs with sequential data, like source code and text.
    - Dependency: RNN layers, including LSTMs, are added to a model.

- Assumption: There is publicly available code smell datasets or free tools to detect code smells.
    - Dependency: I do not need to manually try to find code smells.

- Assumption: If the model works well in detecting a one type of code smell, it should also work well with other code smells.
    - Dependency: Structure of the model is not changed. It stays the same when other code smells are trained.

3.2 Scope and limitations

Scope determines and plans what the prototype will do and what it will not do. In addition, it explains what it will contain and what it will not contain. It sets specific goals for the prototype.

3.2.1 Scope of initial release

The scope of the initial release was just to create a model and train it. It would read dataset from the file and tokenize code samples. Only one or two code smells would be trained to deep learning models from the Java programming language. The model could be saved and loaded to the file, and it could be used to make predictions.

3.2.2 Scope of subsequent releases

The following functions and properties were planned to be released in the future:

- GUI (e.g. website could be done using React.js and TensorFlow.js)
- Read and analyze GitHub repositories.
- Detect more code smells.
- Bigger and more extensive datasets.
- Notify developers when code smells occur.

3.2.3 Limitations and exclusions

The following functions were not planned to be developed in the initial release:

- No graphical user interface.
- It does not find all code smells. Prototype is focusing on finding one or two code smells.
- It may not work on all programming languages, only Java code samples are used in training.

- The prototype does not have to be yet perfect or always right. This is the first experiment.
- Not inclusive and large data. The datasets are remarkably small for this experiment.

3.3 Gathering and preparation of data

The quantity and quality of data are important for the ML algorithms. Even if a trained model were good, it would not work well without proper data.

3.3.1 Gathering

Data gathering manually can be one of the most laborious things when doing research. While doing my research work, I noticed that there are multiple options to gather data:

- **Landfill.** Landfill is an open dataset of code smells with public evaluation. It is a web-based platform for sharing code smell datasets. Five types of code smells were identified from 20 open source software projects. Anyone can contribute to landfill by adding new instances or flagging incorrectly classified instances. (Palomba et al., 2015) Some problems were that source codes were not currently available to download straight from their website and the maintenance of the service did not appear to be very active and it was not open source software.

- **The Qualitas Corpus**. The Qualitas Corpus is large curated collection of open source Java systems. It was created because researchers used different samples which gave different results and they were difficult to compare. It reduces the cost of performing large empirical studies of code and supports comparison. Usually, experiments must be replicated to validate models. (Tempero et al., 2010) The Qualitas Corpus is often used in software engineering research, but it contains Java projects even as old as 2002 (Madeyski and Lewowski, 2020). It is not a code smell dataset and I did not find any publicly available code smell dataset which is based on the Qualitas Corpus.

- **The MLCQ Code Smell dataset**. The MLCQ dataset with nearly 15000 code samples was created by software developers with professional experience who reviewed industry-relevant and contemporary Java open source projects. In general, datasets are often gathered from single individuals whose professional experiences are unknown. They gathered reviews of four code smells. “Blob” and “data class” were on the class-level and “feature envy” and “long method” were on the function-level. The detected code smells are categorized to the four categories by severity: none, minor, major, and critical. The dataset contains links to smelly source code repositories, and it tells the start and end line numbers of code smells. (Madeyski and Lewowski, 2020)

- **Creating own dataset**. Some researchers decide to create new datasets and publish them as part of their research. Good side is that this kind of dataset exactly matches their research need. There are still many drawbacks. Junior developers may not yet know about code smells. Datasets are often not published in suitable form so that they cannot be used for reliable reproduction. In another approach, existing tools analyze selected repositories and you do not need to use professional experience. (Madeyski and Lewowski, 2020)

I decided to use the MLCQ code smell dataset and collect non-smelly code samples myself and combine them to the CSV files. I selected to use the function-level code smells, named the “long method” and the “feature envy”. They can be usually detected just looking at a method. I collected non-smelly code samples from GitHub repositories which had plenty of stars and were popular.

3.3.2 Preparation

I needed to filter the MLCQ code smell dataset by code smell name and severity. I was hoping that high severity code smells were easier to detect by the prototype. There were 78 critical severity “long method” code smells and 24 critical severity “feature envy” code smells. I needed to manually open a link to a repository and copy specific lines to get smelly source code samples. A couple of the links did not work, which slightly reduced the dataset.

I decided to use balanced samples. For example, if I selected to use 20 code smells from the MLCQ dataset, then correspondingly, I would need to collect 20 non-smelly code samples. The "long method" dataset I compiled contains 76 smelly samples and 76 non-smelly samples. The "feature envy" dataset contains 23 smelly samples and 23 non-smelly samples.

3.3.3 Structure

Table 4 shows a sample data structure in the CSV files. The data structure is very simplified, it contains only the identifier, code sample (source code as plain text) and label (labeled to 0 when no code smells are found or 1 when code smells are found).

Table 4. Sample of data structure

id	**codesample**	**label**
…	…	…
2664	"*public static void main(String[] args) {* *System.out.println(""Hello World!"");* *}*"	0
…	…	…

3.4 Requirements

3.4.1 Functional requirements

Functional requirements define those actions and functions that the prototype can execute. The functional requirements, which are ordered by priority, are listed in Table 5.

Table 5. Functional requirements

ID	Description	Priority
REQ-1	Read dataset from the CSV file.	High
REQ-2	Split the dataset to training and validation data.	High
REQ-3	Tokenize code samples.	High
REQ-4	Create, teach, and train a model.	High
REQ-5	Load a model.	High
REQ-6	Make a prediction if code smells or not to at least one type of method-level code smell.	High
REQ-7	Code is written in Python 3 programming language.	Medium
REQ-8	Revision history available.	Medium
REQ-9	Hosted and executable online in Google Colab.	Medium
REQ-10	Uses TensorFlow library.	Medium
REQ-11	The datasets are balanced.	Medium
REQ-12	Training and validation results are visualized.	Medium
REQ-13	Code smells are from the Java programming language.	Medium
REQ-14	There is a GUI.	Low
REQ-15	Create and use self-made tokenizer.	Low

3.4.2 Non-functional requirements

Non-functional requirements are presented in Table 6.

Table 6. Non-functional requirements

Quality attribute	Requirement definition	How?
Scalability and extendibility	The prototype is scalable to train new code smells.	The datasets are on different files. You only need to change the variable name in the source code.
Portability	The prototype can run on all modern browsers.	Google Colab allows you to execute code on Google's cloud servers and no configuration is required.
Modifiability	The prototype should be easily modified and shared.	Google Colab saves all changes to revision history. Everything is automatically saved to the Google Drive folder. You can share your work using the link and give access to others.
Performance	Training the prototype should not take days.	In Google Colab, you do not need high powered computer. You have free access to use Google's graphics processing units (GPUs) or tensor processing units (TPUs).
Reusability	The reusability of the code and the model.	The prototype is an open source project. All the code is commented and high quality. The trained model can be saved and loaded.
Security	Access control.	Only the people who have the right privilege can edit the source code.

3.4.3 Use cases

Use cases shows functional requirements in action. They are presented in Table 7.

Table 7. Use cases

UC-1	**Detect code smell**
Description	The prototype tests if code sample smells.
Actor	Developer.
Normal flow	1. Developer writes some code. 2. Developer copy-paste it to the prototype variable as a string. 3. Developer runs “predict” module of the prototype 4. The prototype prints out a prediction.
Preconditions	The model has been trained in advance or loaded.

UC-2	**Train a model**
Description	The prototype trains a model from a dataset file.
Actor	Maintainer.
Normal flow	1. Data are loaded and shuffled. 2. Data are tokenized by tokenizer. 3. A model is constructed. 4. The model is trained using data. 5. The model is saved to the file.
Preconditions	The CSV file contains the dataset and is hosted online.

4 PROTOTYPE DEVELOPMENT

This part of the thesis shows my development infrastructure and demonstrates code listings that are supported by explanations. The full source code of the prototype is shown in Appendix 1. I used the *Easy Code Formatter* add-in to format pasted code in Microsoft Word.

4.1 Infrastructure

I had previous experience of the ML environment setup. It can be a hassle to install all packages, configure Anaconda package manager and Python virtual environments. Luckily, I found **Google Colaboratory**, "Colab" for short, which allows you to write and execute Python code through a web browser. It is especially well suited to machine learning, data analysis and education. Google Colab is a hosted Jupyter notebook service that requires no setup to use and is free to use. It offers access to free computing resources including GPUs and TPUs. (Google Research, n.d.)

Colab is focusing only on supporting Python and its ecosystem of third-party tools. Colab works with most major browsers, and it is tested with latest versions of Chrome, Firefox, and Safari. Google Colab notebooks are stored in Google Drive, so you can share them easily. Colab adjusts usage limits and hardware availability on the fly, so there are no precise limits and resources. (Google Research, n.d.)

One big advantage of using Google Colab is that you do not need to run all source code as one big package, but you can run the code in small cells and in any order. In addition, outputs are saved, and you do not need to run programs repeatedly.

In addition to Google Colab, I used the following tools, libraries, and packages:

- **TensorFlow**. Google's TensorFlow is a ML framework. It helps implementing models and its biggest benefit for development is abstraction. You do not have to deal with small details of implementing algorithms.

- **Keras**. Keras is a deep learning application programming interface (API). Keras is running on top of the TensorFlow. It provides essential abstraction and building blocks for ML.

- **Scikit-learn**. Scikit-learn is an open source ML library that supports supervised and unsupervised learning. It provides various tools for model fitting, data preprocessing, model selection and evaluation. I used Scikit-learn to shuffle data and splitting data to train and test data.

- **Pandas**. Pandas is a Python library for data analysis and manipulation.

- **NumPy**. NumPy is a Python library, adding support for large, multidimensional arrays and matrices.

- **GitHub**. I mainly used GitHub to host dataset online. It is also suitable to handle version control and to host source code online.

- **Regular expression (regex) and String library.** I used these libraries to preprocess string data. Regex is useful for find and replace operations when modifying strings.

4.2 Importing libraries

All the needed libraries are imported in the beginning of the file, as shown in Listing 1. The line one is specifying a TensorFlow version in Google Colab.

Listing 1. Importing libraries

```
1.  %tensorflow_version 2.x
2.
3. import sklearn
4. import keras
5. import tensorflow as tf
6. import numpy as np
7. import pandas as pd
8. import re
9. import string
10.
11.  from keras.preprocessing import sequence
12.  from keras.preprocessing.text import text_to_word_sequence
13.  from keras.preprocessing.text import Tokenizer
14.  from sklearn.utils import shuffle
15.  from sklearn.model_selection import train_test_split
```

4.3 Loading, shuffling and preprocessing data

I did not find a way to save data permanently to Google Colab so I decided to use GitHub to host my dataset. Now I can easily load my dataset by using the raw web address of the file from GitHub. The first task was to load data to the *X* and *y* variables.

In Listing 2 (lines 6 and 7) there is regex to find all punctuation characters (e.g. *!"#$%&'()*+, -./:;<=>?@[\]^_`{|}~)* and add empty space around them, so tokenizer can separate them to individual tokens. For example, "*AuditLogger.getRoutingTypes(this.addressInfo);*" string would be one token without this process. Regular expression transforms it to "*AuditLogger . getRoutingTypes (this . addressInfo) ;* " and now tokenizer can easily split it to many tokens using space separator.

Data shuffling reduce variance, making sure that model remains general and overfits less. Moreover, data might be in a specific order. For example, when you are splitting data to

training and validation data, you want that datasets contain all the classes, not just one. In line 9, there is *random_state=0* just to obtain the same shuffle every time the script has run.

Listing 2. Loading, shuffling and preprocessing data

```
1. data = pd.read_csv("https://raw.githubusercontent.com/jvirma/code-
   smell-prototype/master/src/long_method.csv", sep=",")
2. data = data[["codesample", "labels"]]
3. X = np.array(data['codesample'])
4. y = np.array(data['labels'])
5.
6. for i in range(len(X)):
7.   X[i] = re.sub('(['+ string.punctuation + '])', r' \1 ', X[i])
8.
9. X, y = shuffle(X,y, random_state=0)
```

4.4 Tokenizing data

I used Keras Tokenizer to tokenize data, as shown in Listing 3. I needed to set some limits, like maximum number of words in vocabulary and maximum length of tokened code sample. The parameter *oov_token* in line 4 adds *UNK* token to word index and it is used to replace out-of-vocabulary words during the *text_to_sequences()* call. I modified all tokens to lower case so that different character sizes do not affect the vocabulary and it remains smaller and more general. NumPy *ravel()*-method returns array as flattened, like [1 2 3, 4 5 6]. Arrays are required in flattened form in the Tokenizer method calls.

Finally, we have *tk.word_index* containing all the words, and *X* containing all the samples as numerical tokens. For example:

- Vocabulary (*tk.word_index*): {'UNK': 1701, '=': 2, '}': 3, '{': 4, 'if': 5, … }
- Tokens (*X*): [[34, 48, 702, 4, 11, 703], [181, 706, 2, 56, 7, 275, 10], …]

Listing 3. Tokenizing data

```
1. max_words = 3000
2. max_len = 3272
3.
4. tk = Tokenizer(oov_token='UNK', num_words=num_words+1, lower=True,
   filters='\n\t')
5. tk.fit_on_texts(X.ravel())
6. tk.word_index = {e:i for e,i in tk.word_index.items() if i <=
   num_words}
7. tk.word_index[tk.oov_token] = num_words + 1
8. X = tk.texts_to_sequences(X.ravel())
```

4.5 Splitting data

Listing 4 shows how to split data using the scikit-learn. The most common and safest way to split training data and testing data is to use the 80/20 rule, often referred to as the Pareto principle. My dataset was so small, so I set this parameter to 70/30, to get more testing data and reduce the variance of the results.

Listing 4. Splitting data

```
1. x_train, x_test, y_train, y_test =
   sklearn.model_selection.train_test_split(X, y, test_size = 0.3)
```

4.6 Adding padding to data

To get all samples to the same length I used Keras padding method, as shown in Listing 5. For example, [1, 2, 3] array would transform to [0, 0, 1, 2, 3] if I had set the parameter *max_len* to 5.

Listing 5. Adding padding

```
1. x_train = sequence.pad_sequences(x_train, max_len)
2. x_test = sequence.pad_sequences(x_test, max_len)
```

4.7 Creating a model

A model is created using many layers, as shown in Listing 6. The *Sequential()*-method groups a linear stack of layers into the model. The first layer is the *Embedding* layer. It turns positive integers (indexes) into dense vectors of fixed size. For example, [[2], [15]] -> [[0.75, 0.2], [0.4, -0.9]]. It can be used only as the first layer in the model.

I tested multiple different models, parameters, and structures. I noticed that creating the model requires plenty of testing for different configurations and inspecting which one works best. I ended up using several *LSTM* layers.

The *Dropout* layer is added because it helps to prevent overfitting. The final layer, the *Dense* layer is just a regular densely connected neural network layer. It gives us the output as one-dimensional and it uses the sigmoid activation function.

Listing 6. Creating a model

```
1. model = tf.keras.Sequential([
2.     tf.keras.layers.Embedding(max_words+2, 16, mask_zero=True,
   input_length=max_len),
3.     tf.keras.layers.LSTM(16, return_sequences=True,
   recurrent_dropout=0.1, dropout=0.1),
4.     tf.keras.layers.LSTM(16, return_sequences=True,
   recurrent_dropout=0.1, dropout=0.1),
5.     tf.keras.layers.LSTM(16, return_sequences=True,
   recurrent_dropout=0.1, dropout=0.1),
6.     tf.keras.layers.LSTM(16, recurrent_dropout=0.1, dropout=0.1),
7.     tf.keras.layers.Dropout(0.2),
8.     tf.keras.layers.Dense(1, activation="sigmoid")
9. ])
10.  model.summary()
```

The summary method in the line 10 prints the summary of my network's structure. Figure 15 shows that output. There you can see all the layers, the shapes, and the amount of trainable params.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 3272, 16)          48032
_________________________________________________________________
lstm (LSTM)                  (None, 3272, 16)          2112
_________________________________________________________________
lstm_1 (LSTM)                (None, 3272, 16)          2112
_________________________________________________________________
lstm_2 (LSTM)                (None, 3272, 16)          2112
_________________________________________________________________
lstm_3 (LSTM)                (None, 16)                2112
_________________________________________________________________
dropout (Dropout)            (None, 16)                0
_________________________________________________________________
dense (Dense)                (None, 1)                 17
=================================================================
Total params: 56,497
Trainable params: 56,497
Non-trainable params: 0
_________________________________________________________________
```

Figure 15. Model summary

4.8 Training a model

I still needed to configure my model with the *model.compile()*-method so that I could train the model with the *model.fit()*-method. Listing 7 shows how I did that. In line 1, I used the binary cross-entropy loss function, which is suitable when there are only two label classes, which are assumed to be 0 or 1 (Keras, n.d.). In line 2, the batch size defines the number of samples that will be propagated through the network before the model is updated. Number of epochs means how many times the algorithm iterates through the entire training dataset. In line 2, the model training is saved to the *history* variable, so the training results, like accuracy and loss, can be visualized later. Figure 16 shows the output of the training phase. Training of the "long method" code smell took about ten minutes.

Listing 7. Training a model

```
1. model.compile(loss="binary_crossentropy",optimizer="rmsprop",metric
   s=['acc'])
2. history = model.fit(x_train, y_train,
   validation_data=(x_test,y_test), epochs=20,batch_size=128)
```

```
Epoch 1/20
1/1 [==============================] - 4s 4s/step - loss: 0.6932 - acc: 0.4906 - val_loss: 0.6935 - val_acc: 0.4783
Epoch 2/20
1/1 [==============================] - 3s 3s/step - loss: 0.6926 - acc: 0.5094 - val_loss: 0.6924 - val_acc: 0.4783
Epoch 3/20
1/1 [==============================] - 3s 3s/step - loss: 0.6908 - acc: 0.5094 - val_loss: 0.6905 - val_acc: 0.4783
Epoch 4/20
1/1 [==============================] - 3s 3s/step - loss: 0.6878 - acc: 0.5094 - val_loss: 0.6850 - val_acc: 0.4783
Epoch 5/20
1/1 [==============================] - 3s 3s/step - loss: 0.6795 - acc: 0.5094 - val_loss: 0.6727 - val_acc: 0.5000
Epoch 6/20
1/1 [==============================] - 3s 3s/step - loss: 0.6573 - acc: 0.5472 - val_loss: 0.6550 - val_acc: 0.5000
Epoch 7/20
1/1 [==============================] - 3s 3s/step - loss: 0.6294 - acc: 0.5189 - val_loss: 0.6006 - val_acc: 0.5870
Epoch 8/20
1/1 [==============================] - 3s 3s/step - loss: 0.5612 - acc: 0.7453 - val_loss: 0.5483 - val_acc: 0.6304
Epoch 9/20
1/1 [==============================] - 3s 3s/step - loss: 0.4881 - acc: 0.7736 - val_loss: 0.4634 - val_acc: 0.9348
Epoch 10/20
1/1 [==============================] - 3s 3s/step - loss: 0.4221 - acc: 0.9811 - val_loss: 0.4331 - val_acc: 0.8478
Epoch 11/20
1/1 [==============================] - 3s 3s/step - loss: 0.3727 - acc: 0.9245 - val_loss: 0.3649 - val_acc: 0.9565
Epoch 12/20
1/1 [==============================] - 3s 3s/step - loss: 0.3131 - acc: 1.0000 - val_loss: 0.3264 - val_acc: 0.9565
Epoch 13/20
1/1 [==============================] - 3s 3s/step - loss: 0.2757 - acc: 1.0000 - val_loss: 0.2958 - val_acc: 0.9565
Epoch 14/20
1/1 [==============================] - 3s 3s/step - loss: 0.2413 - acc: 1.0000 - val_loss: 0.2749 - val_acc: 0.9783
Epoch 15/20
1/1 [==============================] - 3s 3s/step - loss: 0.2366 - acc: 1.0000 - val_loss: 0.2582 - val_acc: 0.9783
Epoch 16/20
1/1 [==============================] - 3s 3s/step - loss: 0.2169 - acc: 1.0000 - val_loss: 0.2434 - val_acc: 0.9783
Epoch 17/20
1/1 [==============================] - 3s 3s/step - loss: 0.1989 - acc: 1.0000 - val_loss: 0.2305 - val_acc: 0.9565
Epoch 18/20
1/1 [==============================] - 3s 3s/step - loss: 0.1945 - acc: 1.0000 - val_loss: 0.2191 - val_acc: 0.9565
Epoch 19/20
1/1 [==============================] - 3s 3s/step - loss: 0.1838 - acc: 1.0000 - val_loss: 0.2087 - val_acc: 0.9783
Epoch 20/20
1/1 [==============================] - 3s 3s/step - loss: 0.1763 - acc: 1.0000 - val_loss: 0.1991 - val_acc: 0.9783
```

Figure 16. Training history

4.9 Saving a model

A model can be saved using the *save()*-method, as shown in Listing 8.

Listing 8. Saving a model

```
1. !mkdir -p saved_model
2. model.save('saved_model/my_model')
```

4.10 Making a prediction

Finally, when the model had already compiled and trained, I could use the model to make predictions about whether any code sample contains code smells. I did the encode function *encode_text()* to perform same tokenizing, preprocessing, and padding operations as earlier. This function takes a code sample as a string input and finally it returns tokens, as shown in Listing 9.

Additionally, I made the predict function *predictSmell()* which prints the prediction results in a well-formed format. For example, the Listing 9 might print following output: "*Model predicts that 'NoSmell' code sample smells in 3.71% probability*".

Listing 9. Making a prediction

```
1. def encode_text(code):
2.   codesample = code.lower();
3.   codesample = re.sub('(['+ string.punctuation + '])', r' \1 ',
   codesample)
4.   tokens = tk.texts_to_sequences([codesample])
5.   tokens = sequence.pad_sequences(tokens, max_len)[0]
6.   return tokens
7.
8. def predictSmell(text, name):
9.   encoded_text = encode_text(text)
10.    pred = np.zeros((1,max_len))
11.    pred[0] = encoded_text
12.    result = model.predict(pred)
13.    print("Model predicts that '" + name + "' code sample smells in
   " + str(np.round(100*result[0][0],2)) + "% probability")
14.
15.  NoSmell = """
16.  void sail() {
17.    LOGGER.info(""The fishing boat is sailing"");
18.  }}
19.  """
20.
21.  predictSmell(NoSmell, "NoSmell")
```

5 RESULTS

This section presents training and validation results of our model for two different code smells. The results were saved to the *history* variable earlier. I used Matplotlib plotting library to visualize the results, as shown in Listing 10.

Listing 10. Plot model training results

```
# Plotting accuracy and loss versus the number of epochs

import matplotlib.pyplot as plt

epochs = range(len(acc))

# Model accuracy

plt.plot(epochs, history.history['acc'], 'ko', label='Training accu
racy')
 plt.plot(epochs, history.history['val_acc'], 'k', label='Validati
on accuracy')
 plt.title('Model accuracy', pad=15, fontsize=14)
 plt.ylabel('Accuracy')
 plt.xlabel('Epoch')
 plt.yticks(np.arange(0, 1.1, 0.1))
 plt.xticks(np.arange(0, len(epochs), 1))
 plt.grid(which='major', axis='both', linestyle=':')
 plt.legend(loc="lower right")
 plt.show()

 # Model loss

 plt.plot(epochs, history.history['loss'], 'ko', label='Training l
oss')
 plt.plot(epochs, history.history['val_loss'], 'k', label='Validat
ion loss')
 plt.title('Model loss', pad=15, fontsize=14)
 plt.ylabel('Loss')
 plt.xlabel('Epoch')
 plt.xticks(np.arange(0, len(epochs), 1))
 plt.grid(which='major', axis='both', linestyle=':')
 plt.legend(loc="lower left")
 plt.show()
```

5.1 “Long method” code smell

This subchapter presents the training and validation results of the model for the “long method” code smell. Training and validation accuracy and loss are represented. In addition, some conclusions are drawn from the results.

5.1.1 Training and validation accuracy

The vertical axis presents the accuracy of the model and the horizontal axis presents the number of epochs. Accuracy is the fraction of predictions that our model predicts right. From Figure 17 can be seen that the model learned 100% accuracy for the training dataset at the 16th epoch. The model was very successful predicting the “long method” code smells from the testing data (~96% accuracy).

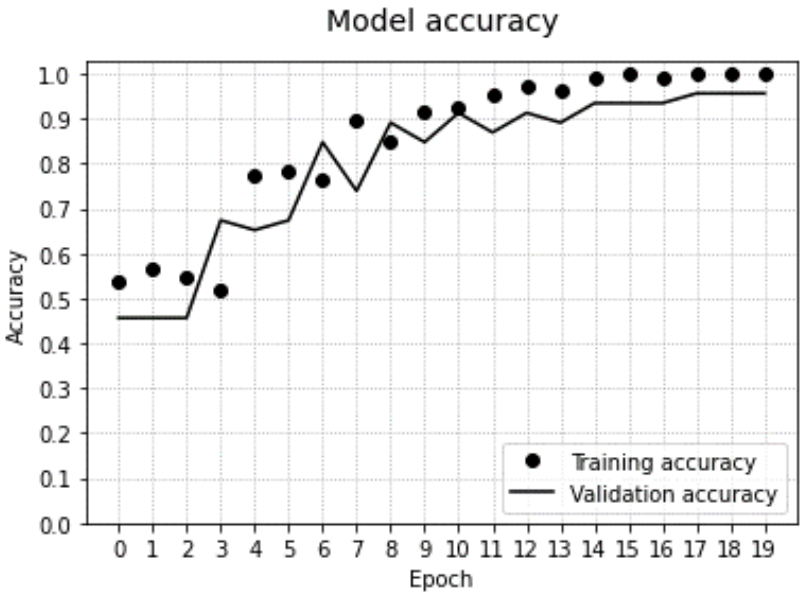

Figure 17. Training and validation accuracy for the “long method” code smell

5.1.2 Training and validation loss

A loss is indicating how bad a model’s prediction was. I used the binary cross-entropy loss function earlier to calculate the loss between true labels and predicted labels. Training loss is the error on the training set of data. Validation loss is the error after running validation set of data through a trained network. If the model were perfect one, a loss would be zero;

otherwise, a loss would be greater. Unlike accuracy, a loss is not a percentage. There are basically three cases to review a loss. (TensorFlow, 2020)

1. **Underfitting**. This occurs when the training loss > the validation loss. This can be caused by a model that might be too complex or too simple.

2. **Overfitting**. This occurs when the training loss < the validation loss. In this case a model is fitting nicely to the training data but not all the validation data. For example, it is not generalizing properly to unseen data. Solutions to this could be to decrease neural network, increase dropout, add weight regularization, or get more training data.

3. **Perfect fitting**. This occurs when training loss == validation loss.

From Figure 18 can be seen that the model is approximately perfectly fitting (training loss == validation loss). The loss could be even smaller if I had used more epochs on model training, because it was still going down in the last epoch. However, the accuracy was already so good that I did not care to decrease the loss anymore, and there was risk to start overfitting to the data.

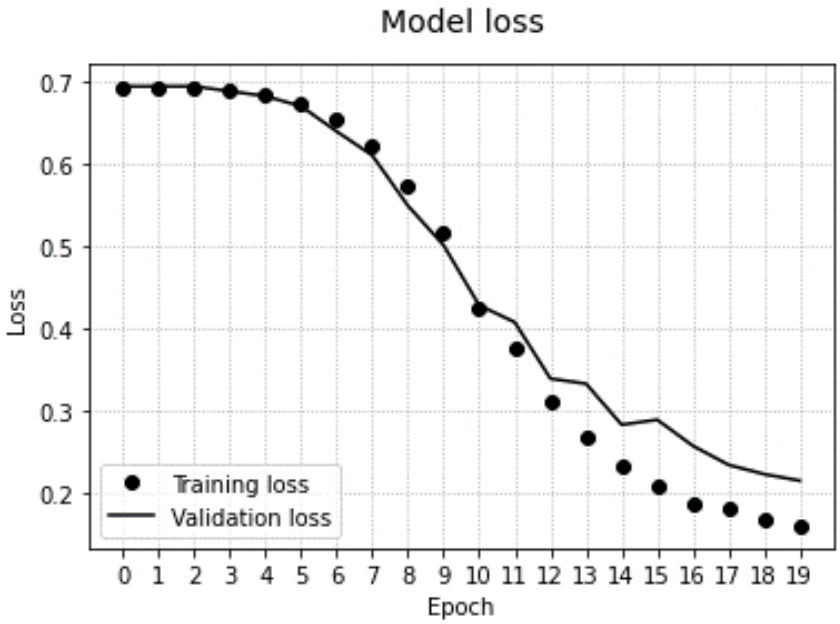

Figure 18. Training and validation loss for the “long method” code smell

5.2 “Feature envy” code smell

This subchapter presents the training and validation results for the “feature envy” code smell. The model and configurations are the same as before.

5.2.1 Training and validation accuracy

Figure 19 presents training and validation accuracy for the “feature envy” code smell. The model learned how to get 100% accuracy for the training dataset. Eventually, the model ended up 87% validation accuracy. It can be concluded that the learning process was successful.

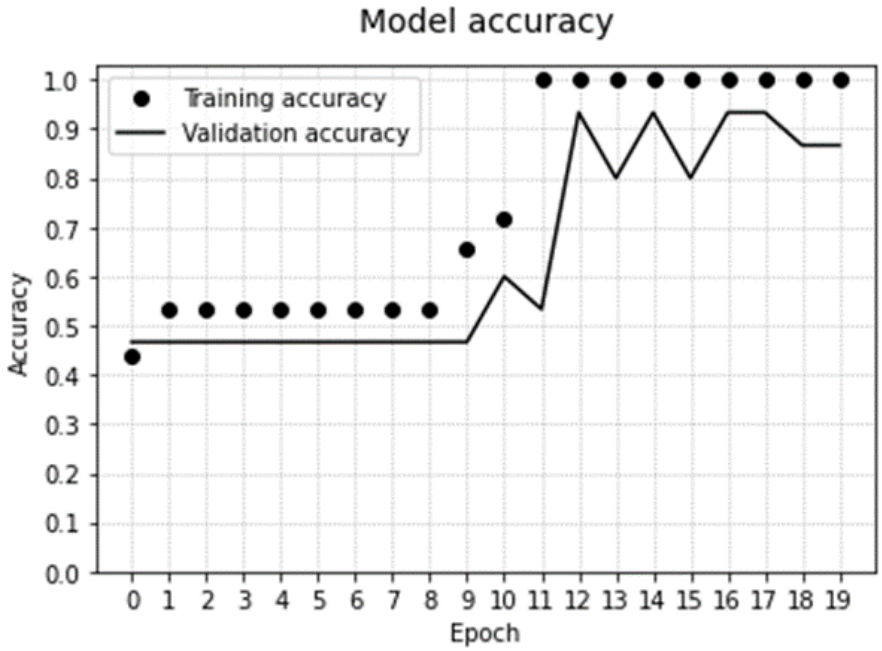

Figure 19. Training and validation accuracy for the “feature envy” code smell

5.2.2 Training and validation loss

Figure 20 represents training and validation loss for the “feature envy” code smell. As we can see, the model started to overfit (training loss < validation loss) at the end of training. The model was not working so well when it needed to predict from unseen data samples. It is possible that more data are required to achieve a better generalizing model.

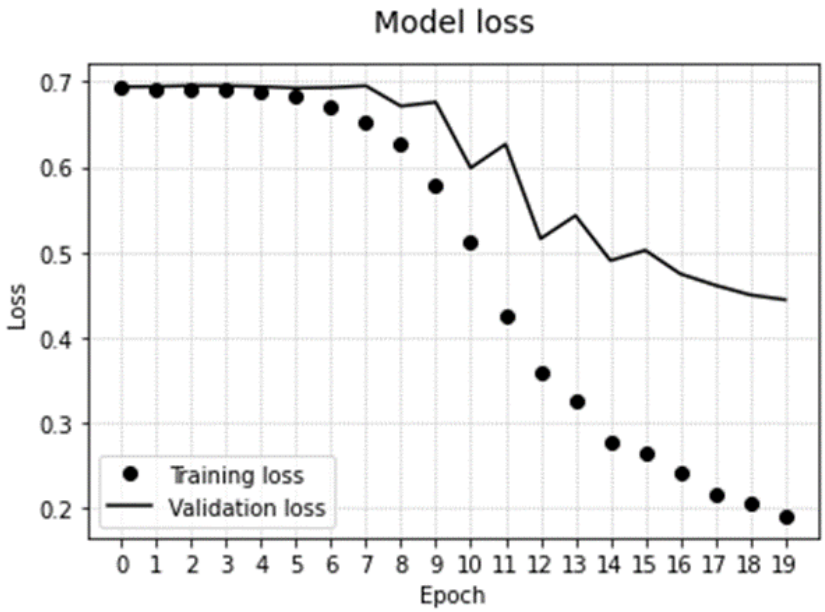

Figure 20. Training and validation loss for the “feature envy” code smell

6 THREATS TO VALIDITY

Reliability and validity are used to evaluate the quality of the research project. It is important to consider the trustworthiness of the results. In what extent the results are true and not biased from my point of view. According to Runeson and Höst (2009), there are four different aspects of validity.

6.1 Construct validity

Construct validity measures the degree to which tools measure the metrics that they are supposed to measure. Relying on the MLCQ dataset may be a threat to the validity because I cannot be sure of the quality of the dataset. However, the MLCQ data were collected from real industry relevant repositories and therefore it should be realistic. In addition, TensorFlow model measurement tools are commonly used and tested, and they should be valid.

6.2 Internal validity

Internal validity refers to the design of the prototype and how confident we can be in the findings. It concerns how outside influences may have an impact on the outcomes. The non-smelly samples I collected may affect the results of the model. The data I collected could be otherwise different from my datasets for the code smells and it could affect the results. Randomization and random selection were used in dataset to improve internal validity. Based on the results, we can conclude that deep learning is feasible to use for detecting at least these two method-level code smells.

6.3 External validity

External validity involves generalizing this study to other people, places, or times. More different code smells and data are required to collect to see how the model really works and how generalizable the results are to the real world. Everything is open source and easily repeated to encourage the replication and building over this research. I made many

assumptions during my research that can affect external validity. The implementation of the models may not be perfect and there are many variables to change.

6.4 Reliability

This aspect concerns in what extent the data and the analysis are dependent on the specific researcher. In other words, how easily results can be reproduced when research is repeated under the same condition. In this case, the collection of non-smelly code samples could differ, but it should not affect the results. Each part of the research study is explained carefully and easily reproducible. Another researcher could choose a different ML algorithm and model to this task or decide to use code metrics data rather than tokens.

7 CONCLUSIONS AND FUTURE DIRECTIONS

According to the literature review, the use of AI is expected to grow massively. This thesis studied AI and its subfields. The goal of the study was to make a prototype utilizing artificial intelligence to detect code smells. I made my own vision plan and implemented it.

The implemented prototype is a code smell detector. It can successfully detect smelly code samples from datasets. I used Python, deep learning, and neural networks in the implementation. The models are capable to learn two code smells, which are the “long method” and “feature envy”.

During the research, I noticed that data collection can become laborious. However, the quality and quantity of data are important for the ML algorithms. In the future, the possibilities of AI will only grow because of the increased amount of digital data created worldwide, its better availability, and better tools. Furthermore, the AI knowledge will grow over time. As the Python ML libraries supported abstraction well, you did not need to know exactly how everything works. While training the models plenty of experimentation with different variables and settings was needed to see what really worked.

The main goal was not to engineer a perfect and ready tool. There are still many ways to enhance the prototype and much remained to be done. The prototype is open source, and everyone can reuse and rework it. However, e.g. GUI, more code smells, and more data are still needed. In the future, it would be interesting to see how it will work with class-level code smells. In addition, it could be tested how it will work with other programming languages, even if the prototype were taught in another language.

The main outcomes from the study are to gain and publish knowledge about how AI can be utilized to solve problems. This thesis presents demonstration about the whole process, starting from the design, and ending to the implementation. The prototype can be modified and used to all kinds of textual sequential data and different problems, like classification.

REFERENCES

Allamanis, M., Barr, E. T., Devanbu, P. and Sutton, S. (2018). *A Survey of Machine Learning for Big Code and Naturalness*. [online] ACM Computing Surveys, Vol. 51, No. 4, Art. 81, pp. 4-17. Available at: https://doi.org/10.1145/3212695

Ammar, H., Abdelmoez, W. and Hamdi, M. S. (2012). *Software Engineering Using Artificial Intelligence Techniques: Current State and Open Problems*. [online] pp. 24. Available at: https://www.researchgate.net/publication/254198356

Biswal, A. (2020) *Recurrent Neural Network Tutorial*. [online] Simplilearn. Available at: https://www.simplilearn.com/tutorials/deep-learning-tutorial/rnn [Accessed 2.7.2020]

Brownlee, J. (2018). *When to Use MLP, CNN, and RNN Neural Networks*. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/when-to-use-mlp-cnn-and-rnn-neural-networks/ [Accessed 7.7.2020]

Brownlee, J. (2019). *14 Different Types of Learning in Machine Learning*. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/types-of-learning-in-machine-learning/ [Accessed 1.7.2020]

Carnes, B. (2020). *Learn how to use TensorFlow 2.0 for machine learning in this MASSIVE free course*. [online] freeCodeCamp. Available at: https://www.freecodecamp.org/news/massive-tensorflow-2-0-free-course/ [Accessed 29.8.2020]

Dam, H. K. (2019). *Artificial intelligence for software engineering*. [online] XRDS: Crossroads, The ACM Magazine for Students. Vol. 25, No. 3, pp. 34-37. Available at: https://doi.org/10.1145/3313117

Dehaghani, S. M. H. and Hajrahimi, N. (2013). *Which Factors Affect Software Projects Maintenance Cost More?* [online] Faculty of Health Services Management and Medical Information Sciences, Isfahan University of Medical Sciences, Iran, pp. 63. Available at: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3610582/pdf/AIM-21-1-15.pdf

Dickson, B. (2017). *What is machine learning?* [online] TechTalks. Available at: https://bdtechtalks.com/2017/08/28/artificial-intelligence-machine-learning-deep-learning/ [Accessed 29.8.2020]

Ďuračík, M., Kršák, E. and Hrkút, P. (2017). *Current Trends in Source Code Analysis, Plagiarism Detection and Issues of Analysis Big Datasets*. [online] Procedia Engineering, Vol. 192, pp. 138. Available at: https://doi.org/10.1016/j.proeng.2017.06.024

Elements of AI. (2019). *A free online introduction to artificial intelligence for non-experts*. [online] Reaktor and University of Helsinki. Available at: https://course.elementsofai.com/ [Accessed 7.7.2020]

Ertel, W. (2011). *Introduction to Artificial Intelligence*. [online] Springer, London, pp. 1, 160-165, 221-223. Available at: https://doi.org/10.1007/978-0-85729-299-5

Existek. (2018). *AI Programming: 5 Most Popular AI Programming Languages*. [online] Existek. Available at: https://existek.com/blog/ai-programming-and-ai-programming-languages/ [Accessed 1.7.2020]

Feldt, R., Neto, F. G. O. and Torkar, R. (2018). Ways of Applying Artificial Intelligence in Software Engineering. In: *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE '18)*. [online] Association for Computing Machinery, New York, USA, pp. 35. Available at: https://doi.org/10.1145/3194104.3194109

Fontana, F. A., Braione, P. and Zanoni, M. (2012). *Automatic detection of bad smells in*

code: An experimental assessment. [online] Journal of Object Technology, Vol. 11, No. 2 (August 2012), pp. 1-4. Available at: https://doi.org/10.5381/jot.2012.11.2.a5

Fontana, F. A., Mäntylä, M. V., Zanoni, M. and Marino, A. (2016). *Comparing and Experimenting Machine Learning Techniques for Code Smell Detection*. [online] Empirical Software Engineering, Vol. 21, pp. 1143–1191 Available at: https://doi.org/10.1007/s10664-015-9378-4

Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, (2nd Edition), pp. 56-60, 85-98

Gartner. (2019a). *Gartner Says AI Augmentation Will Create $2.9 Trillion of Business Value in 2021*. [online] Available at: https://www.gartner.com/en/newsroom/press-releases/2019-08-05-gartner-says-ai-augmentation-will-create-2point9-trillion-of-business-value-in-2021 [Accessed 1.7.2020]

Gartner. (2019b). *Gartner Survey Reveals Leading Organizations Expect to Double the Number of AI Projects in Place Within the Next Year*. [online] Available at: https://www.gartner.com/en/newsroom/press-releases/2019-07-15-gartner-survey-reveals-leading-organizations-expect-t [Accessed 1.7.2020]

Google Research. (n. d.). *Colaboratory - Google*. [online] Google. Available at: https://research.google.com/colaboratory/faq.html [Accessed 12.8.2020]

Gupta, S. and Gupta S. K. (2019). *Natural language processing in mining unstructured data from software repositories: a review*. [online] Sādhanā 44, 244, pp. 1-3. Available at: https://doi.org/10.1007/s12046-019-1223-9

Hadj-Kacem, M. and Bouassida, N. (2018). A Hybrid Approach To Detect Code Smells using Deep Learning. In: *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2018)*. [online] pp. 137-146. Available at: https://www.scitepress.org/Papers/2018/67098/67098.pdf

Highsmith, J. (2002). *Agile Software Development Ecosystems*. [online] Addison Wesley, pp. 83. Available at: http://index-of.co.uk/SE/Addison%20Wesley%20-%20Agile%20Software%20Development%20Ecosystems%20-%20own.pdf

Huilgol, P. (2020). *Quick Introduction to Bag-of-Words (BoW) and TF-IDF for Creating Features from Text*. [online] Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2020/02/quick-introduction-bag-of-words-bow-tf-idf/ [Accessed 12.7.2020]

IBM. (2019). Introducing AI. [online] Coursera. Available at: https://www.coursera.org/learn/introduction-to-ai [Accessed 24.6.2020]

Kasurinen, J. (2020). *Lecture 5: Code smell, refactoring code. Technical debt.* [online] LUT University, CT70A3000 Software Maintenance lecture slides.

Keras. (n.d.) *Probabilistic losses*. [online] Keras. Available at: https://keras.io/api/losses/probabilistic_losses/#binarycrossentropy-class [Accessed 21.9.2020]

Madeyski, L. and Lewowski, T. (2020). MLCQ: Industry-Relevant Code Smell Data Set. In: *Proceedings of the Evaluation and Assessment in Software Engineering (EASE '20)*. [online] Association for Computing Machinery, New York, NY, USA, pp. 342–347. Available at: https://doi.org/10.1145/3383219.3383264

MathWorks. (n.d.). *Unsupervised Learning*. [online] MathWorks. Available at: https://www.mathworks.com/discovery/unsupervised-learning.html [Accessed 1.7.2020]

Nabi, J. (2019). *Recurrent Neural Networks (RNNs)*. [online] Towards Data Science. Available at: https://towardsdatascience.com/recurrent-neural-networks-rnns-3f06d7653a85 [Accessed 29.8.2020]

Pai, A. (2020). *CNN vs. RNN vs. ANN – Analyzing 3 Types of Neural Networks in Deep Learning*. [online] Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2020/02/cnn-vs-rnn-vs-mlp-analyzing-3-types-of-neural-networks-in-deep-learning/ [Accessed 7.7.2020]

Palomba, F., Di Nucci, D., Tufano, M., Bavota, G., Oliveto, R., Poshyvanyk, D. and De Lucia, A. (2015). Landfill: An Open Dataset of Code Smells with Public Evaluation. In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. [online] Florence, pp. 482-483. Available at: http://doi.org/10.1109/MSR.2015.69

Panesar, A. (2019). *Machine Learning and AI for Healthcare*. [online] Apress, Berkeley, CA, pp. 1-2. Available at: https://doi.org/10.1007/978-1-4842-3799-1

Refactoring.Guru. (n.d.). *Code smells*. [online] Refactoring.Guru. Available at: https://refactoring.guru/refactoring/smells [Accessed 28.8.2020]

Petters, K., Tuunanen, T., Rothenberg, M. A. and Chatterjee, S. (2007). *A Design Science Research Methodology for Information Systems Research*. [online] Journal of Management Information Systems, 24:3, pp. 52-56. Available at: https://doi.org/10.2753/MIS0742-1222240302

Rech, J. and Althoff, K-D. (2004). *Artificial Intelligence and Software Engineering: Status and Future Trends*. [online] KI, Vol. 18, pp. 2. Available at: https://www.researchgate.net/publication/220633840

Rumpe, P. (2017). *Agile Modeling with UML*. [online] Springer, Cham, pp. 255-256. Available at: https://doi.org/10.1007/978-3-319-58862-9

Runeson, P. and Höst, M. (2009) *Guidelines for conducting and reporting case study research in software engineering*. [online] Empirical Software Engineering 14, No. 131 pp. 153-154. Available at: https://doi.org/10.1007/s10664-008-9102-8

Sae-Lim, N., Hayashi, S. and Saeki, M. (2017). How Do Developers Select and Prioritize Code Smells? A Preliminary Study. In: *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [online] Shanghai, pp. 1-5. Available at: http://doi.org/10.1109/ICSME.2017.66

Sharma, T., Efstathiou, V., Louridas, P. and Spinnelis, D. (2019). *On the Feasibility of Transfer-learning Code Smells using Deep Learning*. [online] pp. 1-15. Available at: https://arxiv.org/abs/1904.03031

Statista. (2019). *Revenues from the artificial intelligence (AI) market worldwide from 2015 to 2024*. [online] Statista. Available at: https://www.statista.com/statistics/621035/worldwide-artificial-intelligence-market-revenue/ [Accessed 1.7.2020]

Szőke, G., Nagy, C., Fülöp, L. J., Ferenc, R. and Gyimóthy, T. (2015). FaultBuster: An automatic code smell refactoring toolset. In: *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. [online] Bremen, pp. 253-258. Available at: http://doi.org/10.1109/SCAM.2015.7335422

Tempero, E., Anslow, G., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H. and Noble, J. (2010). The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In: *2010 Asia Pacific Software Engineering Conference*. [online] Sydney, pp. 336. Available at: http://doi.org/10.1109/APSEC.2010.46

TensorFlow. (2020). *Overfit and underfit*. [online] TensorFlow. Available at: https://www.tensorflow.org/tutorials/keras/overfit_and_underfit [Accessed 21.8.2020]

TensorFlow. (2019). *Word embeddings*. [online] TensorFlow. Available at: https://www.tensorflow.org/tutorials/text/word_embeddings [Accessed 12.7.2020]

Teradata. (2017). *Survey: State of Artificial Intelligence for Enterprises.* [online] Teradata. Available at: https://www.teradata.com/Blogs/Survey-State-of-Artificial-Intelligence-for-Enterprises [Accessed 1.7.2020]

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A. and Poshyvanyk, D. (2017). When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away). In: *IEEE Transactions on Software Engineering*. [online] Vol. 43, No. 11, pp. 1063-1078. Available at: http://doi.org/10.1109/TSE.2017.2653105

Tzorakoleftherakis, E. *Three Things to Know About Reinforcement Learning*. [online] KDnuggets. Available at: https://www.kdnuggets.com/2019/10/mathworks-reinforcement-learning.html [Accessed 22.9.2020]

Wilson, A. (2019). *A Brief Introduction to Supervised Learning*. [online] Towards Data Science. Available at: https://towardsdatascience.com/a-brief-introduction-to-supervised-learning-54a3e3932590 [Accessed 1.7.2020]

APPENDIX 1. PROTOTYPE

```
1. # This program train model and detects "Long Method" and "Feature E
   nvy" code smells using deep learning (RNN - recurrent neural networ
   k). It encodes code samples to tokens. Code smell data is taken fro
   m MLCQ code smell dataset (Critical severity).
2. # Available to view at https://colab.research.google.com/drive/1KNJ
   n1gzphwtwMj9xQuV9szlziA7OxUFY?usp=sharing
3.
4. %tensorflow_version 2.x
5.
6. import sklearn
7. import keras
8. import tensorflow as tf
9. import numpy as np
10.  import pandas as pd
11.  import re
12.  import string
13.
14.  from keras.preprocessing import sequence
15.  from keras.preprocessing.text import text_to_word_sequence
16.  from keras.preprocessing.text import Tokenizer
17.  from sklearn.utils import shuffle
18.  from sklearn.model_selection import train_test_split
19.
20.  # Load data (feature envy)
21.
22.  data = pd.read_csv("https://raw.githubusercontent.com/jvirma/code
   -smell-prototype/master/src/feature_envy.csv", sep=",")
23.  # Load data for long method
24.  # data = pd.read_csv("https://raw.githubusercontent.com/jvirma/co
   de-smell-prototype/master/src/feature_envy.csv", sep=",")
25.  data = data[["codesample", "labels"]]
26.  X = np.array(data['codesample'])
27.  y = np.array(data['labels'])
28.
29.   # Shuffle and preprocessing
30.
31.  for i in range(len(X)):
32.    X[i] = re.sub('(['+ string.punctuation + '])', r' \1 ', X[i])
33.  X, y = shuffle(X,y, random_state=0)
34.
```

(continues)

APPENDIX 1. PROTOTYPE (continues)

```
35.  # Tokenize data
36.
37.  max_words = 1000
38.  max_len = 1315
39.
40.  tk = Tokenizer(oov_token='UNK', num_words=max_words+1, lower=True
   , filters='\n\t', split=" ")
41.  tk.fit_on_texts(X.ravel())
42.  tk.word_index = {e:i for e,i in tk.word_index.items() if i <= max
   _words}
43.  tk.word_index[tk.oov_token] = max_words + 1
44.  X = tk.texts_to_sequences(X.ravel())
45.
46.  # Printing tokenizing results
47.
48.  print("Vocabulary: " + str(tk.word_index))
49.  print("Vocabulary length: " + str(len(tk.word_index)))
50.  print("Tokens: " + str(X))
51.
52.  # Find out how long is the longest code sample to determine suita
   ble datasize (padding)
53.
54.  def lengths(x):
55.      if isinstance(x,list):
56.          yield len(x)
57.          for y in x:
58.              yield from lengths(y)
59.  print("Longest code sample length: " + str(max(lengths(X))))
60.
61.  # Split data to test and train data
62.  x_train, x_test, y_train, y_test = sklearn.model_selection.train_
   test_split(X, y, test_size = 0.3)
63.
64.  # Padding
65.  x_train = sequence.pad_sequences(x_train, max_len)
66.  x_test = sequence.pad_sequences(x_test, max_len)
```

(continues)

APPENDIX 1. PROTOTYPE (continues)

```
67.  # Create Model
68.  model = tf.keras.Sequential([
69.      tf.keras.layers.Embedding(max_words+2, 16, mask_zero=True, in
   put_length=max_len),
70.      tf.keras.layers.LSTM(16, return_sequences=True, recurrent_dro
   pout=0.1, dropout=0.1),
71.      tf.keras.layers.LSTM(16, return_sequences=True, recurrent_dro
   pout=0.1, dropout=0.1),
72.      tf.keras.layers.LSTM(16, return_sequences=True, recurrent_dro
   pout=0.1, dropout=0.1),
73.      tf.keras.layers.LSTM(16, recurrent_dropout=0.1, dropout=0.1),
74.      tf.keras.layers.Dropout(0.2),
75.      tf.keras.layers.Dense(1, activation="sigmoid")
76.  ])
77.  model.summary()
78.
79.  # Train model
80.  model.compile(loss="binary_crossentropy",optimizer="rmsprop",metr
   ics=['acc'])
81.  #luokitteluun model.compile(optimizer='rmsprop', loss='categorica
   l_crossentropy', metrics=['acc'])
82.  history = model.fit(x_train, y_train, validation_data=(x_test,y_t
   est), epochs=20, batch_size=128)
83.
84.  # Evaluate model
85.  results = model.evaluate(x_test, y_test, batch_size=128)
86.  print("Loss: " + str(results[0]))
87.  print("Accuracy: " + str(np.round(100*results[1],2)) + " %")
88.
89.  # Save model
90.  # !mkdir -p saved_model
91.  # model.save('saved_model/my_model')
92.
93.  def encode_text(code):
94.    codesample = code.lower();
95.    codesample = re.sub('(['+ string.punctuation + '])', r' \1 ', c
   odesample)
96.    tokens = tk.texts_to_sequences([codesample])
97.    tokens = sequence.pad_sequences(tokens, max_len)[0]
98.    return tokens
```

(continues)

APPENDIX 1. PROTOTYPE (continues)

```
# Decode token to words

#reverse_word_map = dict(map(reversed, tk.word_index.items()))
#def sequence_to_text(list_of_indices):
#    words = [reverse_word_map.get(letter) for letter in list_of_indices]
#    return(words)
#my_texts = list(map(sequence_to_text, sequences))
#print(my_texts)

# Use model to predict any code sample. Used two examples

def predictSmell(text, name):
  encoded_text = encode_text(text)
  pred = np.zeros((1,max_len))
  pred[0] = encoded_text
  result = model.predict(pred)
  print("Model predicts that '" + name + "' code sample smells in " + str(np.round(100*result[0][0],2)) + " % probability")

NoSmell = """
void sail() {
  LOGGER.info(""The fishing boat is sailing"");
}}
"""

Smell = """   private static byte[] encodeBase64(byte[] binaryData, boolean isChunked)
public String getProductTitle() {
    var request = HttpRequest.newBuilder()
        .GET()
        .uri(URI.create("http://localhost:51515/information"))
        .build();
    var client = HttpClient.newHttpClient();
    try {
      var httpResponse = client.send(request, HttpResponse.BodyHandlers.ofString());
      return httpResponse.body();
    } catch (IOException ioe) {
      LOGGER.error("IOException Occurred", ioe);
    } catch (InterruptedException ie) {
      LOGGER.error("InterruptedException Occurred", ie);
    }
    return null;
  }
}
"""
predictSmell(NoSmell, "NoSmell")
predictSmell(Smell, "Smell")

```

(continues)

APPENDIX 1. PROTOTYPE (continues)

```
145. # Plotting accuracy and loss versus the number of epochs
146.
147. import matplotlib.pyplot as plt
148.
149. epochs = range(len(acc))
150.
151. # Model accuracy
152.
153. plt.plot(epochs, history.history['acc'], 'ko', label='Training ac
   curacy')
154. plt.plot(epochs, history.history['val_acc'], 'k', label='Validati
   on accuracy')
155. plt.title('Model accuracy', pad=15, fontsize=14)
156. plt.ylabel('Accuracy')
157. plt.xlabel('Epoch')
158. plt.yticks(np.arange(0, 1.1, 0.1))
159. plt.xticks(np.arange(0, len(epochs), 1))
160. plt.grid(which='major', axis='both', linestyle=':')
161. plt.legend(loc="lower right")
162. plt.show()
163.
164. # Model loss
165.
166. plt.plot(epochs, history.history['loss'], 'ko', label='Training l
   oss')
167. plt.plot(epochs, history.history['val_loss'], 'k', label='Validat
   ion loss')
168. plt.title('Model loss', pad=15, fontsize=14)
169. plt.ylabel('Loss')
170. plt.xlabel('Epoch')
171. plt.xticks(np.arange(0, len(epochs), 1))
172. plt.grid(which='major', axis='both', linestyle=':')
173. plt.legend(loc="lower left")
174. plt.show()
```