Modeling code quality using machine intelligence

Erik Kindt

er6425ki-s@student.lu.se

Arvid Malmström

ar2464ma-s@student.lu.se

June 2022

LTH—Lund University
Department of mathematical sciences
Engineering physics

Master's thesis work carried out at Ingka Group Digital.

Supervisors: Carina Geldhauser, carina.geldhauser@math.lth.se
Dan Wahlin, dan.wahlin1@ingka.ikea.com

Examiner: Gabriele Brüll, gabriele.brull@math.lth.se

Abstract

For any company with a software development branch, one of the most important aspects is to write maintainable, understandable, high-quality source code. This will result in fewer work hours to refactor the code if changes are needed. Therefore it's expensive to keep working with source code of poor quality. The question is, how to measure the quality of source code?

The writers of this thesis approached the problem using machine learning. Firstly two literature studies were conducted, one concerning usable software metrics and one concerning usable machine learning algorithms. Secondly, a large, external, labeled data base was set up and the models were trained on said data. There were different final models set up. Two to focus on large projects consisting of hundreds of files and two to focus on singular stand-alone files. After examining the models, the ones with the lowest RMSEs on the test set were used to compare the predictions from the models with the opinions of experienced developers.

The final models were built using the algorithms artificial neural network and random forest which both gave similar results. The models for specific files were tested for both front end and back end files where four files in each were ranked according to the final score from the model. The rank was then compared to a ranking of the same files by experienced software developers. The results showed for front end files, that the model agreed with the developers ranking to a large extent. For back end files, there was a wider discrepancy between the opinions of the developers and the predictions made by the model.

The final models were also able to evaluate a large project over time, to evaluate the overall code quality, which also was one thought application of the model. When a project that was rewritten a year ago was continuously evaluated, the code quality improved which was the expected result according to developers in the project.

Keywords: Machine Learning, Artificial Intelligence, Software Development, Software Metrics

Acknowledgements

First of all, the writers of this thesis would like direct a big thank you to the entire team at Ingka Group Digital. Especially to our supervisor Dan Wahlin for always supporting us with both theoretical and practical issues, as well as always being invested in the writing of the thesis. Also a big thank you to Magnus Pettersson, who is the responsible manager, for allowing us the opportunity to write our thesis at Ingka Group Digital.

Secondly we would also like to direct a big thank you to our supervisor at LTH, Carina Geldhauser. Thank you for always being someone to turn to regarding any machine learning algorithm and for giving us necessary constructive criticism regarding the structure of the thesis.

Finally we would also like to thank the team members of the team at Ingka who helped us evaluating our models.

Contents

Abbreviations

ANN	Artificial Neural Network
CC	Cyclomatic Complexity
CD	Code Duplication
CQ	Code Quality
JS	JavaScript
LOC	Lines Of Code
MAE	Mean Absolute Error
ML	Machine Learning
MSE	Mean Squared Error
RDF	Random Forest
RMSE	Root Mean Squared Error
TS	TypeScript

1 Introduction

When developing software, it's always important to keep track of the quality of the code. A correct management and measuring of the code will lead to more easily understood, easily modified and more efficient projects for any organization, which will lead to a more economic and sustainable code standard. The question of what "good" code actually is varies between companies as well as between the individual developers. However, there is consensus about different ways of measuring and analysing code, as well as which of these metrics plays a bigger role when developing software.

This thesis will try to build a model using machine intelligence algorithms and known metrics to evaluate the quality of written source code. The idea is to build a suitable model using well established Machine Learning (ML) algorithms while also using widely accepted software metrics. The model will take software metrics as inputs and produces a score or ranking as an output to give an indication on the weather or not the Code Quality (CQ) is up to the desired standard.

1.1 Previous work

This thesis should be regarded as a continuation or alternation of the master thesis 'Identification of Technical Debt in Code using Software Metrics' by Erica Schillström and Dan Wahlin from 2021. The work from last year consisted of a deep dive in the theory regarding the concept of *technical debt*. After this work had been conducted through a literature study as well as interviews with developers at Ingka, a model to measure technical debt was constructed. The thesis can be found here.

Since the model were measuring technical debt, the writers of this thesis were to build a similar model, but measuring code quality rather than technical debt. The model used last were also a rather simple model, with a sum of inputs with respective weights. The weights were set up in accordance with the opinions of software developers at the company. This year, the model were going to measure code quality rather than technical debt and were to be built using ML-algorithms. Instead of using the opinion of software developers to construct the models, their opinion would be used to validate the models after it was built.

1.2 Purpose

The purpose of the thesis is to investigate and understand how well software code quality can be measured and modeled using machine intelligence. The final model is planned to both be used during development to get an idea if the projects CQ is improving during development, as well as the use of comparing different projects to each other with regards to CQ.

1.3 Research Questions

To properly investigate the purpose of the thesis, three research questions were formed.

- **Q1.** Is it possible to construct a ML model to evaluate a company's software code quality?
- **Q2.** In the specified data set, which ML algorithms achieve the best performance?
- **Q3.** What are the possible usages of such a model?

The thesis is supposed to find answers to these three research questions.

1.4 Scope & limitations

The scope of the thesis is that it will only consist of supervised learning algorithms with continuous outputs. It will not consider unsupervised learning algorithms nor supervised learning algorithms with discrete outputs. The thesis will only feature well established, commercially used software metrics found in regularly used, commercial code analysis tools such as Designite, SonarQube, Embold etc.

1.5 Outline

The thesis begins with a short introduction and then swiftly moves on to give a theoretical background in section 2. This section has two purposes, to partly give the reader an introduction to software metrics, but also to provide the reader with the mathematical and theoretical knowledge regarding the used machine learning algorithms. It also describes the data set used for training the model in depth. In chapter 3 the methods used when writing the thesis are described while chapter 4 presents the results and evaluation from the models. Chapter 5 will focus on discussing the results of the previous chapter, as well as answering the questions presented above in section 1.3. Chapter 6 will wrap up the thesis in conclusions while the final chapter 7 will present possible future work.

1.6 Ingka Group Digital

The thesis is written in collaboration with Ingka Group Digital and will here on forward be named as simply Ingka. The company is part of the IKEA franchise. The writers of this thesis have been working closely to two software development teams at Ingka.

The software development in these teams is structured in an agile way, while implementing a development process in accordance with the Scrum model.

The product the team is currently working on is an application to create membership or finding existing memberships. The application is for touchscreens inside the stores around the world. Other products focused on are arranging a lottery for different markets and a function to cancel an existing membership. The products depend on data from other teams, so they have to collaborate with other teams in the company. There are some employees in the team working with data science, but it was seen as an interesting opportunity since there was an expressed desire from the team to incorporate more data science in general and machine intelligence specifically into the team's pipeline.

The purpose of the models is to be able to evaluate code quality of software code at the two concerned team at first and implemented in their pipeline. After this has been successfully done, there might be possibilities to incorporate the model in other parts of Ingka as well.

2 Theoretical background

In this chapter, a theoretical background will be provided. It will cover the data set used to train and test the models. It also will cover both descriptions of software metrics used in the models and machine learning algorithms used in the practical part of the thesis. Especially the Random forest algorithm and the Artificial Neural Network will be used in the thesis. Some other useful terms when discussing CQ will also be described.

2.1 The Qscored data set

The *Qscored* data set is an open source data set which is available to the public[1] and the base of a paper, Sharma and Kessentini (2021), which the following section is written in accordance with. It was created in July 2020 and is continuously extended. At the point of writing this thesis, it contains analysis of more than 109 000 different projects from Github. The projects in Qscored is written in the programming languages Java or C#. This analysis consists of measures of 26 software metrics as well as a ”quality score” or a *Qscore* connected to each project. This score indicates the CQ of each project. The relationships is inverted, the lower the score, the higher the CQ.

The constructor of the data set, professor T Sharma, has built the data set using the Designite code analysis tool, which he has also built. This tool calculates a collection of software metrics for Java and C# code.

One thing to note regarding the data set, is that it contains data for the same project, but at different points in time. Therefore there are a lot of data points which have roughly the same metrics.

The Qscore of each project is calculated based on the detected code smells in each field, according to equation 2.1 below,

$$Qscore = \frac{\omega_a \times ASD + \omega_d \times DSD + \omega_i \times ISD}{\omega_a + \omega_d + \omega_i}, \tag{2.1}$$

where ASD is the architecture smell density, DSD is the design smell density and ISD is the implementation smell density. ω_a, ω_d and ω_i is the corresponding weights. The concept of code smells is described in section 2.2. The reader should note that **the lower the score, the higher the code**

[1]Available to download as of May 2022 at https://zenodo.org/record/4468361#.Ym-YT_NBxJU

quality.

To determine the values of these weights, professor Sharma conducted a survey where 31 experienced software developers gave their opinion on which type of code smell were the most important to direct their work on for a high quality code. The results were that the majority of the software developers would direct 50 % their time towards architectural issues, 30 % of their time towards design issues and 20 % of their time towards implementation issues. Therefore the weight were set as $\omega_a = 0.5$, $\omega_d = 0.3$ and $\omega_i = 0.2$. The score calculated with equation 2.1 is the labels for the models in this thesis and the metrics in the data set the features.

Since the Qscore provided by the data set will act as labels in the constructed model will act as ground truth. This will of course introduce a bias towards the model used in Qscored, but it was decided that it was a necessary sacrifice to be able to use supervised learning. This would enable the writing of the thesis and hopefully shed new light on the possibility of modeling CQ using ML.

2.2 Code smell

The term *code smell* were popularized by Fowler in 1999. It is used to describe any signs in the source code that there might be a hidden underlying problem with the structure of the code. These could be problematic choices of implementation or general design of the program. For example, according to Paiva et al., a for-loop where the iteration index is not increased is a type of code smell. The presence of these code smells suggest that the code need to be *refactored*, meaning changing the code to improve the internal structure and integrity of the program.

Fowler introduced 22 different types of code smells in his book *Refactoring: Improving the design of existing code*. This list includes duplicated code, long methods, large classes and so on. In the next subsection some of the commonly identified code smells are listed, to give the reader an introduction to the concept of code smells. The most important is the following smells according to *Object-oriented metrics in practice*:

2.2.1 God class

According to Marinescu (2006), a *god class* is when one class has most of the intelligence of the system. The class then performs too much of the work by itself and rely on data from other classes. The god classes are often large and complex. This can slow down a software system.

To detect a god class there are three main components which is:

- Access data from simpler classes.
- Big classes that are long and complex.
- Low cohesion between the methods in that class.

Those are shown in the following figure:

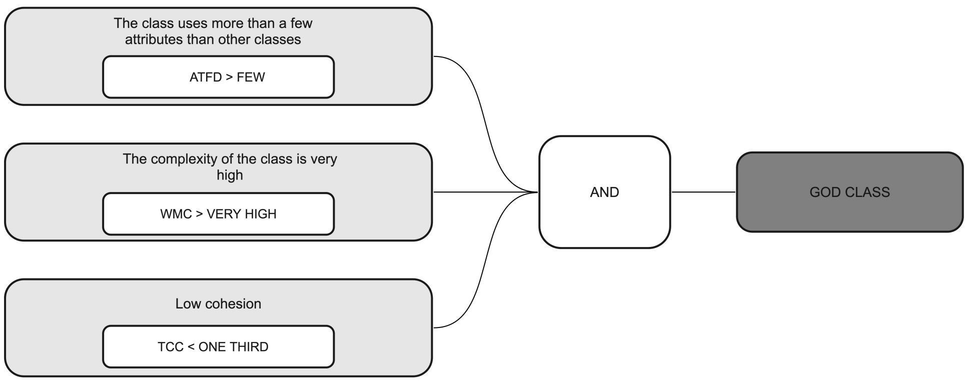

The class uses more than a few attributes than other classes
ATFD > FEW
The complexity of the class is very high
WMC > VERY HIGH
Low cohesion
TCC < ONE THIRD
AND
GOD CLASS

Figure 1: How to detect a god class (AFTD - access to foreign data).

If the class uses many foreign attributes, then the ATFD will be high and be more probable to become a god class. The functional complexity is measured by a weighted method count.

2.2.2 Brain method

Another often identified code smell is the so called *brain method*. A brain method is a software method which centralizes the intelligence of a class and often takes the shape of a method which is very long, complex and complicated to understand. This often leads to that the brain method is challenging to maintain for software developers, demanding that a lot of hours to refactor if needed according to Vidal et al. (2018).

2.2.3 Brain class

Olbrich et al. (2010) defines a *brain class* as, in similarity with a god class, a class which contains a lot of intricacy and complexity. The main difference however is that a brain class doesn't use much data from external or outside classes, not in the same degree as a god class does.

2.2.4 Feature Envy

Feature envy is described by Paiva et al. (2017) as the process when a method in an object which is mainly using attributes from other objects. This can for example be if an object evokes get-methods on other type of objects to then use this data to do calculations within the original object class according to Fowler (1999). This points to the fact that the method should be placed in the class which uses is the most.

2.2.5 Data Class

A *data class* is a class in the code which doesn't contain enough complexity. They mainly store data and other classes are heavily dependent on the data class, according to Fontana et al. (2015). They often contain a lot of attributes, but lack logical methods to go with them. It exposes the data to other classes through accessor methods.

2.3 Maintainability & Testability

The *maintainability* describes the easiness or difficulty with which a program can be modified. According to Visser et al. (2016), this includes e.g. if the code is easy to understand, if it's robust or how easy it is to find and correct bugs. These changes often occur both before and after the release of the software product according to Lewis and Bassetti (2000).

Another important aspect of developing software is to be able to test the program through out the development-process, this is described as a programs *testability*. In an easily testable program, it's easy to find possible problems in the source code. If a program is not easily testable, it might be required to go through a process of refactoring and redesigning to be able to find these bugs according to Freedman (1991).

2.4 Front and back end

The following part is based on an article by Almuttairi (n.d.).

Developers often make a difference between front end and back end development. Front end is the part of a website or application that interacts with the user. Front end includes things like text colour, graphs and tables etc which are things the user experience directly. Also the design, structure and behavior of what is seen on a web page or application is designed by a front end developer. Front end files also needs to be responsive and not slow which also is a task for a front end developer. Front end files are usually written in HTML, CSS and JavaScript.

The back end developer is focused on the server side of the website or application. In the back end, the data is stored, updated and arranged for the front end side to use. It is the part of the website or application that the user do not interact with. Back end files are usually written in PHP, C++, Java, Python and JavaScript.

2.5 A selection of other software metrics

When discussing quality of code, this is often subject to different opinions. Different developers and companies have different approaches and priorities to design their code and programs. However, to concretize the code, it's often useful to introduce metrics of the code. This can for example be the size of the project or the number of lines in a file. In this section, software metrics used in the model will be introduced. These metrics are typically not regarded as code smells, but other seperate metrics of the code.

2.5.1 Lines Of Code (LOC)

This section is written in accordance with Pfleeger (2015). One rather simple, but still powerful software metric is to count the LOC in the project. This is a metric used to measure the size of the project. LOC is an often used metric by different developers and organizations, but one thing to note regarding the metric is that it matters whats written in each line of code. For example one line might just be blank or contain a comment, while another contains an entire while loop. The complexity is greater in the second alternative, but when counting LOC they both simply result in one line of code. The metric LOC is still a powerful tool, but it's important to be aware of what it actually says about the code.

2.5.2 Number of functions/methods

The software metric number of methods is very similar to lines of code as it is also a size metric. The metric count the number of methods in a file or a project.

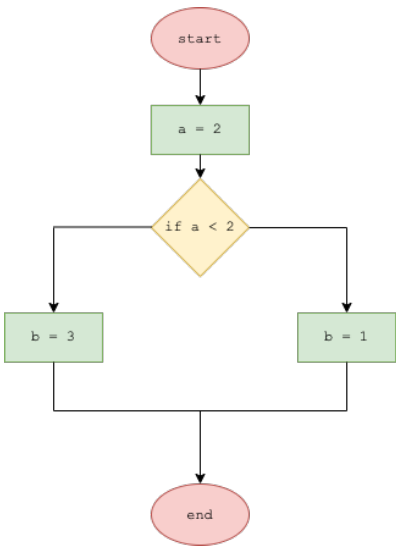
2.5.3 Code Duplication (CD)

Another important metric that can be used to measure the quality of code is if the code is often duplicated. These are blocks of code that could probably be refactored e.g. turned into a method or saving the results from the previous time the block was executed. This is often a result when a developer simply uses copy and paste on a block of code. One problem with code duplication is that if a bug is found in one of the code blocks, and it's modified, then the second code block will also have be updated, something that might be forgotten according to Visser et al. (2016).

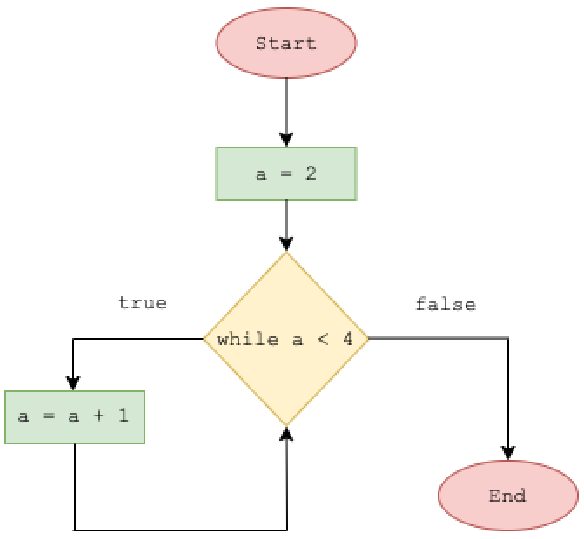
In the used data set, the level of code duplication within a project is calculated with a index-based algorithm suggested by Hummel et al. (2010). This index maps sequences of normalized statements to their occurrences. This algorithm is outside the scope of the thesis, but will ultimately result in a percentage of how much code is duplicated.

2.5.4 Control Flow Chart

When examining a written software programs complexity, a *flow chart* is often used to described the algorithm. This flow chart will display every possible path through the program. It will prove a useful tool when working with certain software metrics e.g. *Cyclomatic complexity.* Two examples of control flow charts are displayed in figure 2 below.



(a) Flowchart of an if statement.



(b) Flowchart of a while statement.

Figure 2: Flowchart of different two statements.

2.5.5 Cyclomatic complexity

One way to describe the complexity of the code is using McCabe's Cyclomatic Complexity (CC). This is combined with flow charts of the code and is defined by equation 2.2 below, according to Pfleeger (2015).

$$V(f) = E - N + 2 \tag{2.2}$$

In equation (2.2) V is is the CC, f is the selected flowchart, E is the number of edges is the chart and N is the number of nodes in the chart.

Looking back at the flow charts from section 2.5.4, for example, the flowchart in figure 2a would have $N = 6$ and $E = 6$ which would result in a CC of $V(f) = 6-6+2 = 2$. Since CC measures the number of linearly independent paths through f it is easy to realize there are two in figure 2a.

2.6 Machine intelligence algorithms

The thesis will now move on to target machine learning rather than data science. This will be done by describing some foundations of machine learning, rigorously introduce the algorithms used and also swiftly discuss the ways to evaluate and compare different models with each other.

2.7 Supervised learning

Supervised learning is one of the foundations of machine intelligence. The algorithm is trained on *labeled* data, meaning each sample of the training set has a label connected to it. This means that every combination of parameter values in the training set also has a predetermined output. The models is modifying its parameters to match the output of the training set. The idea is that the trained model will later be able to predict an output on unseen data as input, according to Marsland (2009). This can be done for several types of problem, e.g. regression and classification tasks. Supervised learning stands in contrast with *unsupervised learning* which doesn't use any labels in it's algorithms. Unsupervised learning are often use for tasks like clustering or dimensionality reduction. In the following section some supervised ML-algorithms will be introduced.

2.8 ANN networks

When discussing predictive supervised machine learning algorithms, one of the most powerful tools is the Artificial Neural Network (ANN). The ANN is a name for several algorithms which replicates the neurons in the human brain. Figure 3 shows the algorithm of one simple neuron, or a *node*, using three inputs.

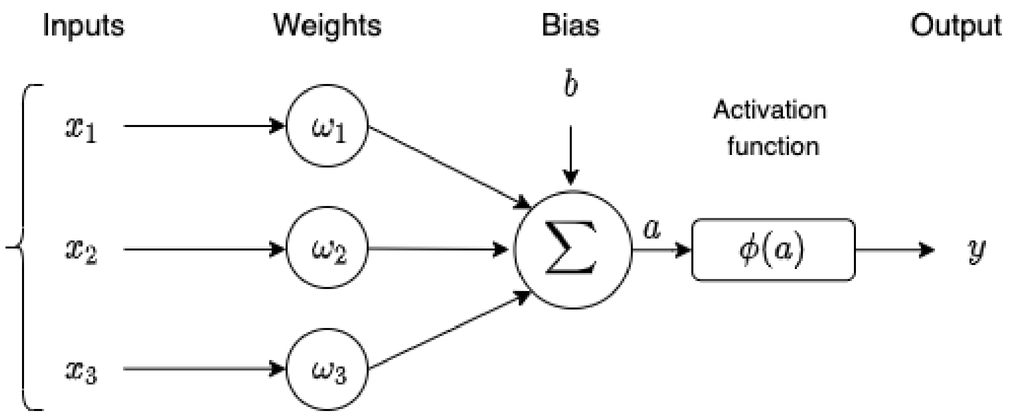


Figure 3: A simple neuron using three inputs. Present in the figure is also the weights, the bias, the activation function as well as the output.

The output from a neuron is calculated according to equation 2.3 below,

$$\begin{aligned} a &= \sum_{k}^{N} \omega_k x_k + b \\ y &= \phi(a) \end{aligned} \tag{2.3}$$

where ω_k is weight k, x_k is input k, b is the bias and ϕ is called the *activation function*. This activation function often contains a threshold behaviour for some level of input and will determine which output in each node in the network will produce. There are several popular activation functions, e.g.

- Linear - $\phi(a) = a$
- Heaviside - $\phi(a) = \theta(a)$
- ReLU - $\phi(a) = max(0, a)$

which are also shown in figure 4.

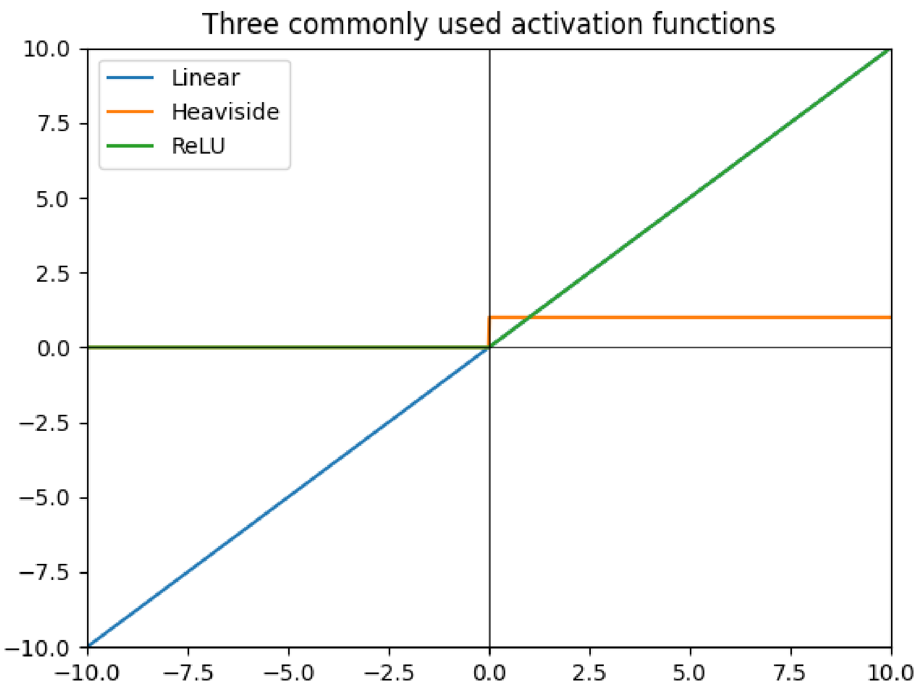


Figure 4: Three commonly used activation functions, linear, heaviside and ReLU.

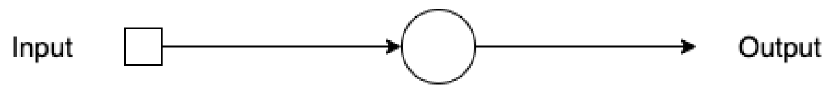
The heaviside activation function is often used for binary classification and also when using gradient descent for back propagation, the derivation of the heaviside is non differentiable at $x = 0$ and 0 elsewhere. So when updating the weights, gradient descent will not make progress.

The linear activation function always produces a constant output when using gradient descent. Another downside of always using a linear activation function is that the output will also be linear and a function of the first layer. So a non-linear relationship will not be picked up if this activation function is chosen.

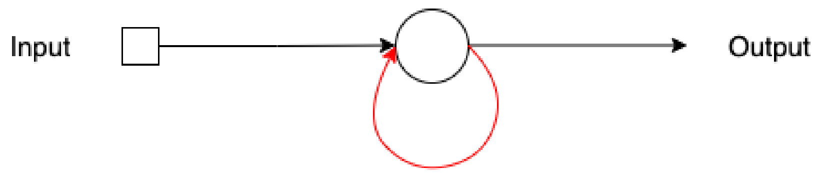
One upside of using the ReLU is that it is good at handling vanishing gradients. Since it is non-linear it can also capture non-linear relationships between the input and the output. ReLU is also rather fast since it is always equal to zero when $x < 0$. However the ReLU can cause some nodes to be saturated and will thereafter always produce negative results, according to Edén Ohlsson (2021).

The structure of the nodes like the ones in figure 3 is called the networks *architecture*. This includes both how many layers of nodes is used in the network, as well as how many nodes there are in each respective layer. The two major ways of structuring a network are called *feed forward*- and *feed backward* architecture. The feed forward network doesn't have any closed signal loops and the inputs are always processed forward through different layers.

The feedback or *recurrent network* does however, as the name suggest, introduce the option to include feedback connections according to Edén Ohlsson (2021). A simple diagram of the two architectures are displayed in figure 5.



(a) A simple feed forward network.



(b) A simple recurrent network.

Figure 5: Two simple neural networks.

A feed forward network is often suited for e.g. regression types of problems. The goal of the network is often to generate output close to the targets. A recurrent network is often suited for dealing with sequence data, such as text data, speech data or numerical time series according to Edén Ohlsson (2021).

When building neural networks for regression types of problems, one of the most commonly used type of layer is the *fully connected* (or dense in Keras) layer. This means that each of the nodes in the layers is connected to all the nodes in the previous layer. Two of these fully connected layers is displayed in figure 6 below.

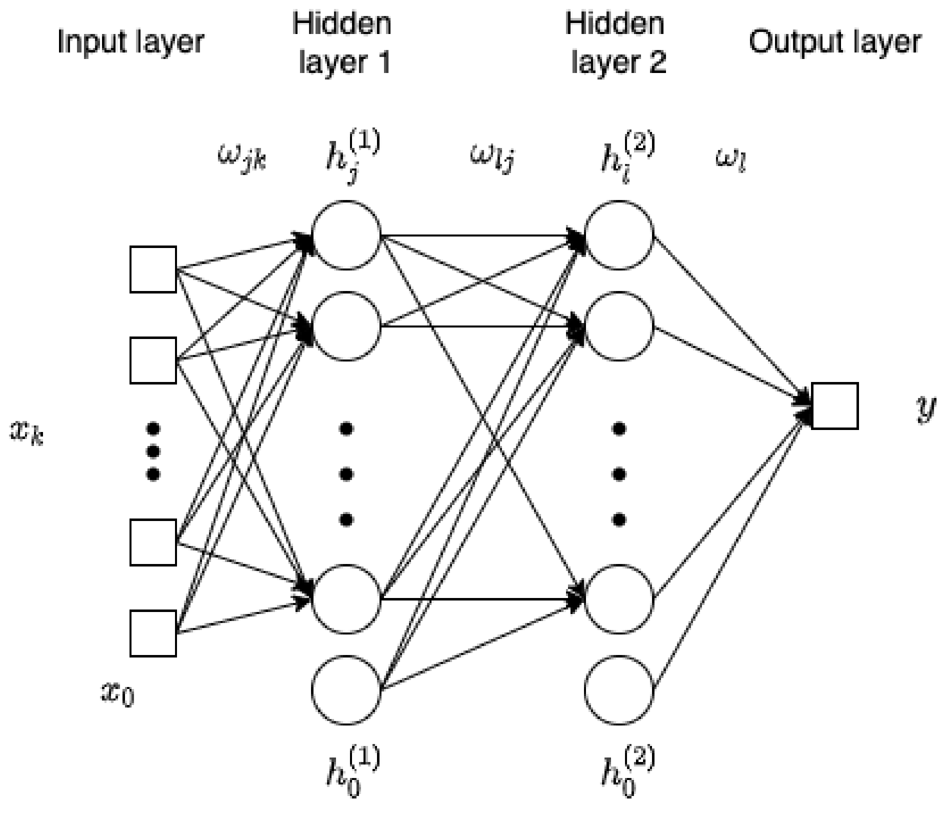


Figure 6: Two fully connected hidden layers. The two biases are shown in the bottom of the screen.

In figure 6, x_k represent input k, ω_{jk} weight on connection from input k to node j, $h_j^{(1)}$ node j in hidden layer 1 and so on. x_0, $h_0^{(1)}$ and $h_0^{(2)}$ is the biases. The usage of this biases is to shift the activation function and thereby moving the threshold. The output y from the network above would be calculated according to

$$y = \phi_0 \left(\sum_{l=1} \omega_l \phi_h^{(2)} \left(\sum_{j=1} \omega_{lj} \phi_h^{(1)} \left(\sum_{k=1} \omega_{jk} x_k + \omega_{j0} x_0 \right) + \omega_{l0} h_0^{(1)} \right) + \omega_0 h_0^{(2)} \right). \tag{2.4}$$

In equation (2.4), ϕ_0 is the activation function of the output node and $\phi_h^{(1)}$ and $\phi_h^{(2)}$ is the output functions of the two hidden layers.

When adding more layers, equation 2.4 expands rapidly. But it's still not the most complex equation, it's rather just sums of weights, inputs and biases.

The reader should note the term *hidden layer* in figure 6. A hidden layer indicates a layer in between the input and the output. In figure 6 there are two hidden layers present. It is possible to build a network just connecting the input to the output, such as the simple perceptron, but it is more common to use several hidden layers according to Edén Ohlsson (2021).

Another commonly used layer is the *dropout* layer. This is added to the network to avoid overfitting. It means that there is a chance for weights to be set to zero for nodes and there by dropping the node. The dropping of these nodes are applied randomly with parameter p, which is the probability to keep a node. These nodes are only temporarily dropped, but returned the next time the training is done, as stated by Edén Ohlsson (2021). Figure 7 shows a network before and after dropout is applied.

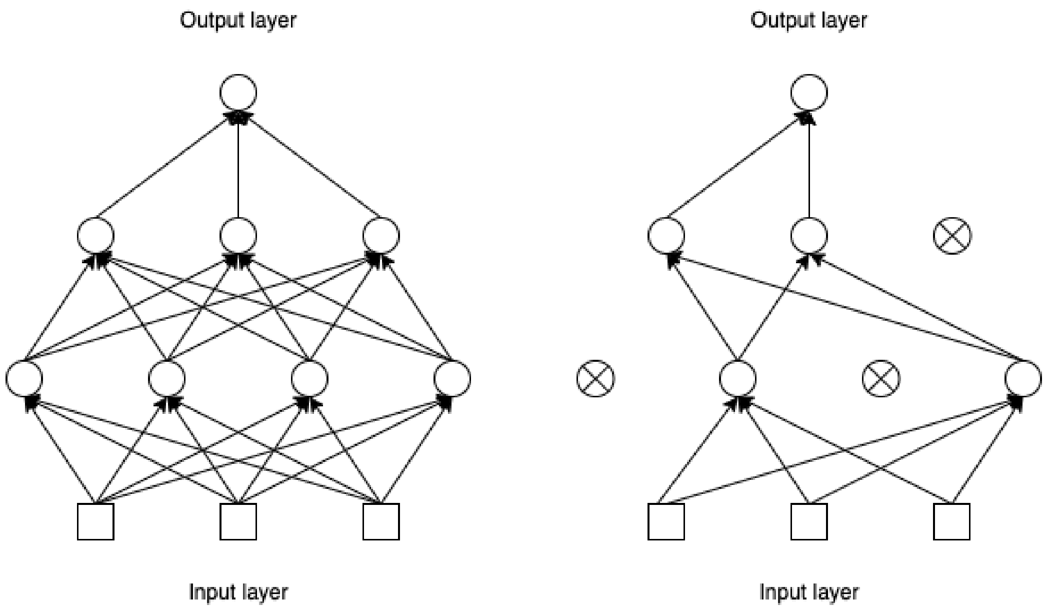


Figure 7: A neural network before and after dropout have been applied. The nodes marked with a cross have been dropped.

2.9 Loss functions, optimizers, learning rates, batch sizes and epochs

For the network to know what the purpose of the algorithm is, it is necessary to introduce a *loss function*. This function, often denoted by $\mathcal{L}(\boldsymbol{\omega})$, which is the function to be minimized during training and thereby optimizing the algorithm. The weights and biases defined in equation (2.3) is represented with the symbol $\boldsymbol{\omega}$ and the aim is to find the $\boldsymbol{\omega}$ that solves the problem as well as possible. For a regression type of problem with continuous output it is common to use the Mean Squared Error (MSE) loss function which is stated in equation 2.5 below.

$$\mathcal{L}(\boldsymbol{\omega}) = \frac{1}{2N}\sum_{n=1}^{N}(y_n - d_n)^2. \tag{2.5}$$

In equation (2.5) $\boldsymbol{\omega}$ is the weights, N is the number of samples, $y_n = y(\boldsymbol{\omega}, \boldsymbol{x_n})$ is the output for sample n and d_n is the corresponding target. Note that the factor $\frac{1}{2}$ could be discarded, it won't affect the result of the optimization according to Edén Ohlsson (2021).

Since this loss function is to be minimized, this has now turned into a optimization problem. The goal is to find weights $\boldsymbol{\omega}$ which minimizes the loss in equation (2.5). It's possible to differentiate all of the losses and activation functions according to equation (2.6)

$$\frac{\partial \mathcal{L}}{\partial \omega_k} = \sum_n \frac{\partial \mathcal{L}}{\partial y_n} \cdot \frac{\partial y_n}{\partial a_n} \cdot \frac{\partial a_n}{\partial \omega_k} = \sum_n \frac{\partial \mathcal{L}}{\partial y_n}\phi'(a_n)x_{nk}, \tag{2.6}$$

where ω_k is the k-th input weight, a_n is the input to the output function ϕ at sample n. One strategy, or *optimizer* is to update the weights in the network according to $\Delta\boldsymbol{\omega} = -\eta\nabla_{\boldsymbol{\omega}}\mathcal{L}$, where η is the *learning rate* which can be arbitrarily chosen. This is known as the *gradient descent algorithm* or *gradient descent optimizer* according to Edén Ohlsson (2021).

As seen above the choice of optimizer used to train the network can play a big part in the training of the network. This is the algorithm used to minimize the loss function. When dealing with regression problems, it is often common to use the the optimizer known as *root mean squared propagation* or RMSProp for short.

RMSProp is closely related to the gradient descent optimizer. It's designed to increase the speed of the process when finding the minimum of the loss function by reducing the number of function evaluations according to Goodfellow et al. (2016). It uses a weighted, moving average of the calculated gradients, discarding early extreme gradients. The algorithm is shown in pseudocode in algorithm 1.

Algorithm 1 The RMSProp algorithm.

```
**Requires**: Global learning rate $\eta$
**Requires**: Initial parameters $\omega$
**Requires**: Small constant $\delta$ for numerical stability
Initialize accumulation variables $\mathbf{r} = 0$
while Stopping requirement not fulfilled: do
    Sample a minibatch $m$ examples from training set $\{x^{(1)}, ..., x^{(m)}\}$ with corresponding targets $y^{(i)}$.
    Compute the gradient $\mathbf{g} \leftarrow \frac{1}{m}\nabla_\omega \sum_i \mathcal{L}(f(x^i;\omega), y^{(i)})$
    Accumulate squared gradient $\mathbf{r} \leftarrow \rho\mathbf{r} + (1-\rho)\mathbf{g} \odot \mathbf{g}$.
    Compute parameter update; $\Delta\omega = \frac{-\eta}{\sqrt{\delta + \mathbf{r}}} \odot \mathbf{g}$ .
    Apply update: $\omega \leftarrow \omega + \Delta\omega$.
end while
```

In algorithm 1, a new hyperparameter is introduced in ρ which controls the length of the moving average. δ is a small constant, often $\delta = 10^{-7}$, and $\odot$ represent elementwise multiplication between the gradients $\mathbf{g}$. RMSProp has been shown to work well for neural network, both practically and effectively. It is one of the most common optimizers for neural networks according to Goodfellow et al. (2016).

As seen above, one of the hyperparameters that need to be decided on is the *learning rate*. This indicated how much of the modification of the weights should be included in the new neural network. This is included in all neural networks, weather it's feed forward, recurrent or convolutional networks. This learning rate can be arbitrarily chosen but is often, according to Edén Ohlsson (2021), selected as a small number, e.g. $\eta = 0.01$. However when deciding the learning rate it's a lot of trial and error since it depends on the data and the architecture of the network. A to high learning rate leads to an unstable loss reduction while a to high leaning rate makes the loss reduction slow according to Edén Ohlsson (2021).

Goodfellow et al. (2016) also states that learning rates goes hand in hand with the number of *epochs* used when training the network. An epoch means an update of all the weights in the entire network when all of the training points has been passed through the network. However since this often is very computational-heavy it is preferred to send that data through in *batches.* The sizes of these batches can be arbitrarily chosen and depends on the size of the data set.

2.10 Decision trees

This section about decision trees is based on section 16 in the book Murphy (2013).

CART stand for *classification and regression trees* and are commonly known as decision trees. The basics of a decision tree is that the input space is recursively divided where each region contains it's own model.

The following example explains shows the basics of a decision tree.

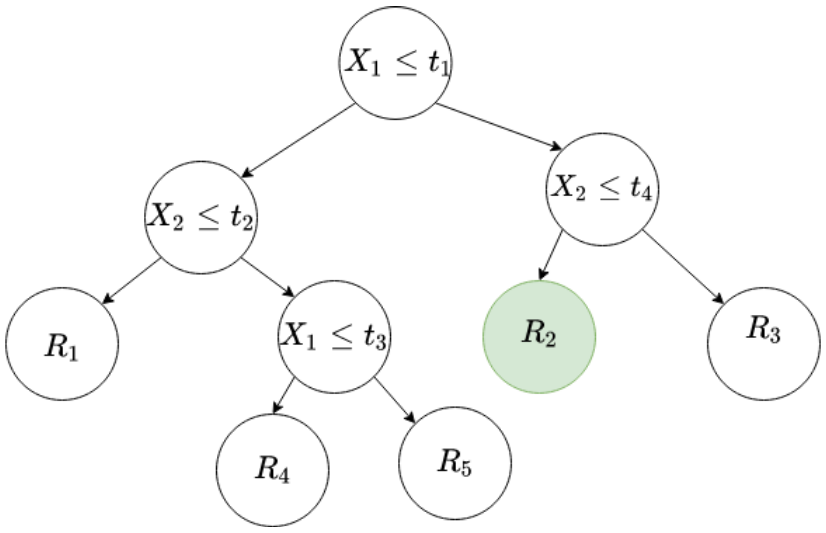


Figure 8: Basic decision tree where R_2 is the final output.

In the first step, if x_1 is less or equal to the threshold t_1 then it will follow the left leaf and to the right if it is bigger than t_1. With continuous splits, each input ends up in a leaf (in this example R_{1-5}). This will split the input space into five regions. The prediction can then be written as

$$T(x;\Theta) = \mathbb{E}[y|\mathbf{x}] = \sum_{i=1}^{N} \omega_i \mathbb{I}(\mathbf{x} \in R_i) = \sum_{i=1}^{N} \omega_i \theta(\mathbf{x}; \mathbf{v}_i) \tag{2.7}$$

where ω_i is the mean response in region R_m and $\mathbf{v}_i$ is the variables chosen to split on and their threshold value. Θ is the parameters $\Theta = \{R_i, \omega_i\}$.

2.10.1 Building a decision tree

The most common way to build a tree is to start with a single leaf which is called the root. The leaf is assigned a value from a majority vote in the training set. This procedure is then done iterativly for each leaf and each split is examine in terms of a gain. Of all the possible splits (or the option not to split at all) the best one is chosen with the one that maximizes the gain.

The problem to partition data in an optimal way is NP-complete and therefore a greedy approach to find the optimal solution for the partition. There are many algorithms that could be used to build a decision tree. Some of those are ID3, C4.5 and CART. For all of those it's common to use the following algorithm to find the locally optimal MLE(maximum likelihood estimation).

Algorithm 2 Recursive procedure to grow a decision tree.

```
1. function fitTree(node,$\mathcal{D}$,depth);
2. node.prediction = mean($y_i, i \in \mathcal{D}$)//or class label distribution;
3. ($j^*, t^*, \mathcal{D}_{\mathcal{L}}, \mathcal{D}_{\mathcal{R}}$) then
4. IF not worthSplitting(depth,cost,$\mathcal{D}_{\mathcal{L}}, \mathcal{D}_{\mathcal{R}}$)) then
5.   return node
6. else
7.   node.test = $\lambda$x.$x_{j^*} < t^*$ //anonymous function;
8.   node.left = fitTree(node,$\mathcal{D}_{\mathcal{L}}$,depth+1)
9.   node.right = fitTree(node,$\mathcal{D}_{\mathcal{R}}$,depth+1)
10   return node
```

Where $\mathcal{D}$ is a tree or a subtree, t is a threshold for for a specific node, $\mathcal{T}$ is the set of thresholds, x is the input and y is the response.

Algorithm 2 chooses the best features and value to split on according to:

$$(j^*, t^*) = \arg \min_{j \in 1,\dots,D} \min_{t \in \mathcal{T}_j} cost(\{\mathbf{x}_i, y_i; x_{ij} \leq t\}) + cost(\{\mathbf{x}_i, y_i; x_{ij} > t\}) \quad (2.8)$$

Where (j^*, t^*) are the optimal split feature and the threshold. $\mathcal{T}_j$ is all the thresholds for a one feature(duplicates would not be counted.).

There are many ways to check weather the split is worth doing or not.

- If the cost reduction is too small.
- If the tree has reached the maximum depth.

- If either $\mathcal{D}_{\mathcal{L}}$ or $\mathcal{D}_{\mathcal{R}}$ is homogeneous enough, then there is no more splitting of that set.
- If numbers of samples in $\mathcal{D}_{\mathcal{L}}$ or $\mathcal{D}_{\mathcal{R}}$ is too small.

For regression problems, which is the problem in this thesis, the cost function is defined as

$$cost(\mathcal{D}) = \sum_{i \in \mathcal{D}} (y_i - \bar{y})^2 \tag{2.9}$$

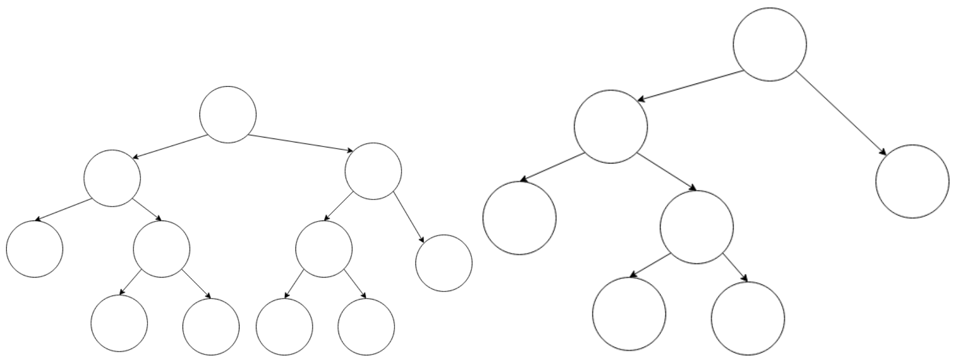
where $\bar{y}$ is the mean response.

2.11 Pruning

The section about pruning is based on the section 3 in Breiman et al. (1984).

One way to avoid overfitting in decision trees is to use make use of *pruning*. When just running algorithm 2, the trees tends to be overfitted to the training data.

Pruning will often remove nodes and synapses in the decision which is shown in the following figure.



(a) Decision tree before pruning. (b) Decision tree after pruning.

Figure 9: How pruning changes a decision tree.

According to Breiman et al. (1984), one commonly used algorithm is cost complexity pruning.

The cost complexity measure for a subtree is defined as:

$$R_\alpha(T) = R(T) + \alpha|\tilde{T}| \tag{2.10}$$

where the complexity parameter $\alpha \geq 0$ is a real number and $|\tilde{T}|$ is the number of terminal nodes.

Using equation 2.10, for each value of α, the aim is to find the subtree $T(\alpha) \leq T_{max}$ which minimizes equation 2.11:

$$R_{\alpha}(T(\alpha)) = \min_{T \leq T_{max}} R_{\alpha}(T) \tag{2.11}$$

For a small α, the penalty will for a lot of terminal nodes be small but for a large alpha, the penalty will be big. Therefore with increasing α, there will be smaller and smaller amount of terminal nodes.

2.12 Bootstrapping

The following section will introduce the technique *bootstrap* to measure the accuracy of a model. Section 2.13 will focus on how to use bootstrapping to improve a model prediction.

Suppose a model is to be fitted to a training set $\mathbf{X} = x_1, x_2, ..., x_N$. Then a random draw with replacement is done on the data set, producing B different bootstrap datasets with the same size as the original data set. The model is then fitted to each of the bootstrap datasets and the model can be examined to study the models behaviour to all the samples according to Hastie et al. (2001).

Say the prediction of the model is denoted as $\hat{f}(\mathbf{X})$ for any input point. Any aspects of the distribution of $\hat{f}(\mathbf{X})$ can the be examined, therefore giving information how the model behaves to different datasets with slight variation.

The next step would be to use bootstrapping to estimate a prediction error. If $\hat{f}^{*b}(x_i)$ is the prediction at x_i, derived from the model that were fitted to the b-th bootstrap dataset, the estimate is given by equation (2.12) below,

$$\hat{Err}_{boot} = \frac{1}{B}\frac{1}{N}\sum_{b=1}^{B}\sum_{i=1}^{N}\mathcal{L}(y_i, \hat{f}^{*b}(x_i)), \tag{2.12}$$

where B is the number of bootstrapped samples, N is the number of data points in the original dataset, y_i is the models predicted value and $\mathcal{L}$ is the chosen loss function.

This approach, however, might introduce another problem. The reason this approach does not provide a good estimate, is because in this case the bootstrapped samples are acting as training samples and the original training set is the test set. Since all of these samples contains overlapping samples, overfitting can occur and make prediction errors lower than they actually would be when predicting unseen data.

One way to work around this is to define the $\hat{Err}$ differently. Defining it as equation (2.13), leaves out the bootstrap samples b that does *not* contain observation i, denoted by C^{-i}. $|C^{-i}|$ is the number of such samples.

$$\hat{Err} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{|C^{-i}|}\sum_{b\in C^{-i}}\mathcal{L}(y_i, \hat{f}^{*b}(x_i)). \tag{2.13}$$

The method described above is known as the *leave out* bootstrap. This procedure means that the bootstrapped samples will not any longer have the same size as the original dataset. The leave out method approach does solve the earlier stated overfit error, but still does have a problem with imposing a bias concerning the size of the training set. The bias is introduced because of the non-distinct observations in the bootstrap samples that originates from the sampling with replacement according to Hastie et al. (2001). However bootstrapping will still be useful when used through bagging in training of the random forest. The leave out bootstrap method is displayed in figure 10.

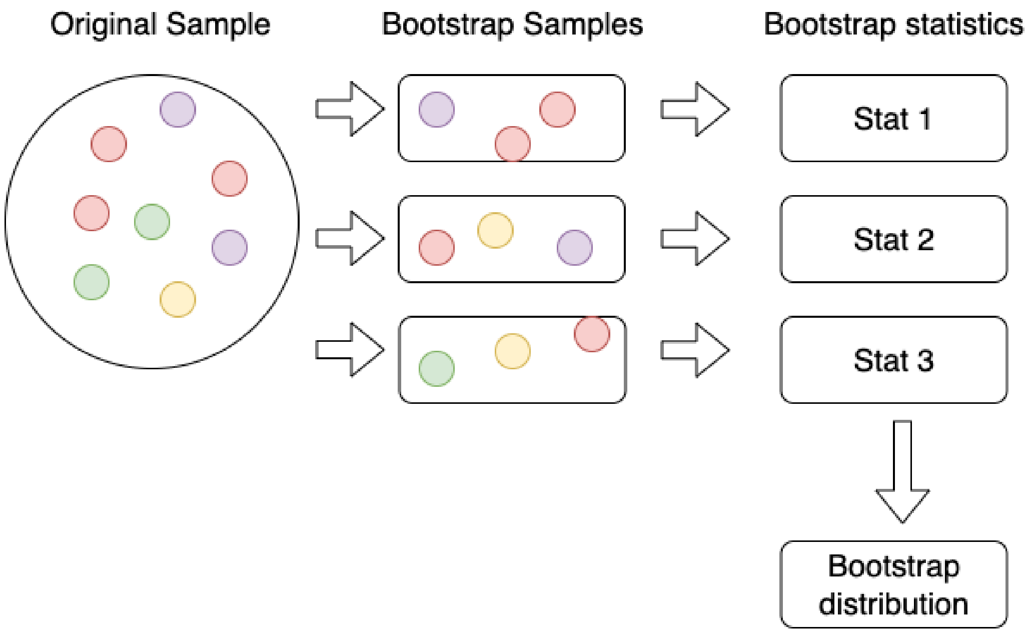


Figure 10: The leave out bootstrap algorithm.

2.13 Bagging

Bagging or *Bootstrap aggregating* is, as stated earlier, a way to improve the predictions of a model. The following section will describe the bagging method for the regression problem.

Suppose a model is to be fitted to a training set $\mathbf{X} = x_1, x_2, ..., x_N$. The goal is that the model will make prediction $\hat{f}(x_i)$ at input x_i. The algorithm creates B different bootstrap samples and the model is fitted to each of these samples and makes a prediction $\hat{f}^{*b}(x)$ for the b-th sample. The bagging estimate is then defined according to equation (2.14) Hastie et al. (2001).

$$\hat{f}^{*b} = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{*b}(x). \tag{2.14}$$

This isn't equal to the true bagging estimate, but rather a Monte Carlo estimate approaching the true estimate as $B \to \infty$.

2.14 The Random Forest (RDF) algorithm

Bagging or *bootstrap aggregation* is a technique used to reduce the variance of a predictor. The technique is described in depth in section 2.13. The bagging work especially well for high variance, low bias predictors such as the decision tree. This technique can be used to train several decision trees on slightly different training sets, to reduce the single decision trees tendency to overfit. This is known as the *random forest* algorithm. The algorithm constructs an *ensemble* of decision trees which is a collection of several decision trees. Therefore the algorithm can combine the predictions of several different decision trees to reduce the risk of overfitting that often occurs with the single tree. The final prediction consists of an average over all the predictions from all the trees in the forest. The algorithm is depicted below in figure 11.

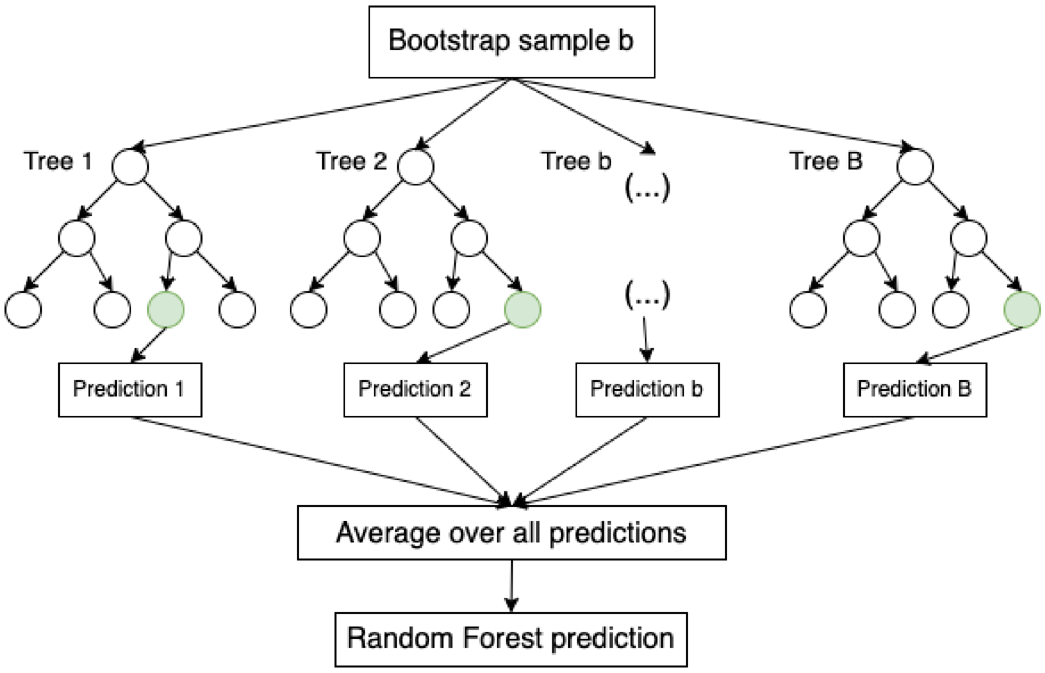


Figure 11: The random forest algorithm consisting of N trees.

Note in figure 11 the term "Bootstrap sample b". The algorithm doesn't train all B trees on the whole of the training set, instead it randomly samples the training set using bootstrapping, as described earlier in section 2.12.

The algorithm is described in pseudocode below as algorithm 3 in accordance with Hastie et al. (2001).

Algorithm 3 The random forest algorithm for regression

```
1. For b = 1 to B:
    a) Draw a bootstrap sample $\mathbf{X}^*$ of size $\mathbf{N}$ from the training data
    b) Grow a random-forest tree $T_b$ to the bootstrapped recursively
    data, by repeating the following steps for each terminal node of
    the tree, until the minimum node size $n_{min}$ is reached.
        i. Select m variables at random from the p variables.
        ii. Pick the best variable/split-point among the m.
        iii. Split the node into two daughter nodes.
2. Output the ensemble of trees $\{T_b\}_1^B$.

To make a prediction at a new point $x$:

$\hat{f}_{rf}^B(x) = \frac{1}{B}\sum_{b=1}^B T_b(x)$
```

One of the downsides of using the random forest algorithm is that the easy interpretability of the single decision tree is lost. The decision tree is often easy to understand, as seen in figure 8, but with the random forest it rapidly gets more difficult. The forest often consist of several hundreds of trees, depending on the size of the data set. However the sacrifice in interpretability is often repaid in higher accuracy since the risk of overfitting of the model is reduced accoording to Murphy (2012).

2.15 Gradient boosting

Another variant of decision trees is the gradient boosting algorithm. Gradient boosting is a technique that uses an ensemble of weak learners (in this case decision trees) to predict on a regression problem.

A decision tree can be defined as equation (2.7). A boosted tree is in this case a sum of such trees.

$$f_m(x) = \sum_{m=1}^{M} T(x; \Theta_m). \tag{2.15}$$

Where T is the tree, x is the input and Θ_i is the parameters R_i and ω_i according to equation (2.7).

The difference between gradient boosting and random forest is how the trees in equation (2.15) are built. When a boosting technique is used, then the residual for the sum of the previous trees are calculated and from the residual a new tree is added. This makes boosting an adaptive technique where the new tree is dependant of the previous error. For each new tree, the following has to be solved:

$$\tilde{\Theta}_m = \arg\min_{\Theta_m} \sum_{i=1}^{N} \mathcal{L}(y_i, f_{m-1} + T(x_i; \Theta_m)). \tag{2.16}$$

Equation (2.16) shows that the new constrains and regions of the next tree is given by the current model by minimizing a loss function. The best tree is then the decision tree that best predicts the residuals $y_i - f_{m-1}(x_i)$.

As equation (2.16) shows, a loss function is needed which should be differentiable. The following are the most common loss functions too use.

Table 1: The most common loss functions and their gradients for regression.

Loss function $\mathcal{L}(y_i, f(x_i))$	$-\dfrac{\partial \mathcal{L}(y_i, f(x_i))}{\partial f(x_i)}$
$\frac{1}{2}[y_i - f(x_i)]^2$	$y_i - f(x_i)$
$\lvert y_i - f(x_i)\rvert$	$sign[y_i - f(x_i)]$
Huber	$y_i - f(x_i)$ for $\lvert y_i - f(x_i)\rvert \leq \delta_m$ $\delta_m sign[y_i - f(x_i)]$ for $\lvert y_i - f(x_i)\rvert > \delta_m$ where $\delta_m = \alpha$th-quantile$\{\lvert y_i - f(x_i)\rvert\}$

Usually the default option is in table 1 is squared error to calculate the residual. Hastie et al. (2001).

For gradient boosting the difference compared to equation (2.16) is that instead of only fitting to the residual, instead the tree should be fitted to the negative gradient. This gives the new equation for squared errors as a loss function:

$$\tilde{\Theta}_m = \arg\min_{\Theta} \sum_{i=1}^{N} (-g_{im} - T(x_i; \Theta))^2, \tag{2.17}$$

where $g_i = \dfrac{\partial \mathcal{L}(y_i, f(x_i))}{\partial f(x_i)}$ is the residual of the loss function.

This results in algorithm 4, described in pseudocode below:

Algorithm 4 The gradient boosting algorithm for regression

```
1. Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} \mathcal{L}(y_i, \gamma)$.
2. For m = 1 to M:
    a) For $i = 1, 2, ..., N$ compute $r_{im} = -\left[\frac{\partial \mathcal{L}(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}$
    b) Fit a regression tree to the targets $r_{im}$ giving terminal regions $R_{jm}$,$j = 1, 2, ..., J_m$.
    c) For $j = 1, 2, ..., J_m$ compute
       $\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} \mathcal{L}(y_i, f_{m-1}(x_i) + \gamma)$.
    d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Output $\hat{f}(x) = f_M(x)$
```

2.16 Ridge regression

In the simple linear regression, the least square error is used to calculate the minimum between the actual and the predicted values. The goal is to minimize this difference, to make the best prediction on new data points. Another way to approach a regression problem is by using so called *shrinkage methods.* These methods introduces a penalty function to each coefficient according to Hastie et al. (2001).

The *ridge regression* approach imposes a penalty to the coefficients depending on their respective sizes. It penalizes a residual sum of squares , where the ridge estimate is calculated according to equation (2.18),

$$\hat{\beta}^{ridge} = \arg\min_{\beta} \Big\{ \sum_{i=1}^{N} (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \Big\}, \tag{2.18}$$

where y_i is the true values for each training case n, x_{ij} is the input for training case n, β_j is the jth coefficient and λ is the shrinkage parameter. The β_0 is often known as the intercept. Equation (2.18) can also be expressed as equation (2.19). The $\lambda \geq 0$ controls the amount of shrinkage, how largely the coefficient are to be penalized. The larger the λ, the greater shrinkage are performed on the coefficients. The coefficients are therefore shrunk towards zero with increasing λ.

$$\begin{aligned} \hat{\beta}^{ridge} &= \arg\min_{\beta} \sum_{i=1}^{N} (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2. \\ \mathit{subject\ to} & \sum_{j=1}^{p} \beta_j^2 \leq t. \end{aligned} \tag{2.19}$$

One thing to note in equations (2.18) and (2.19) is that the intercept β_0 is left out of the penalty term. This is because adding this would result in the ridge estimate being dependent on the chosen origin for the result interval Y, since the inputs are not *equivariant* when subject to scaling. This means that the adding of a a constant c to each of the targets y_i would not simply shift the prediction by the same amount c. Therefore one should usually scale the inputs before applying the penalty term and solving equation (2.18). This approach is also used in neural networks, but goes by another name, *weight decay* according to Hastie et al. (2001).

The constraints in the two equations (2.18) and (2.19) can be visualized in figure 12. The figure shows the two-parameter case. On the axes there are the two parameters β_1 & β_2. The red ellipses are the least square error function. The blue circle is the constraints show in equation (2.18). Note that the constraints takes the shape of a disc (in the two-parameter case). This means that there are no room for any parameters to be shrunk all the way down to 0.

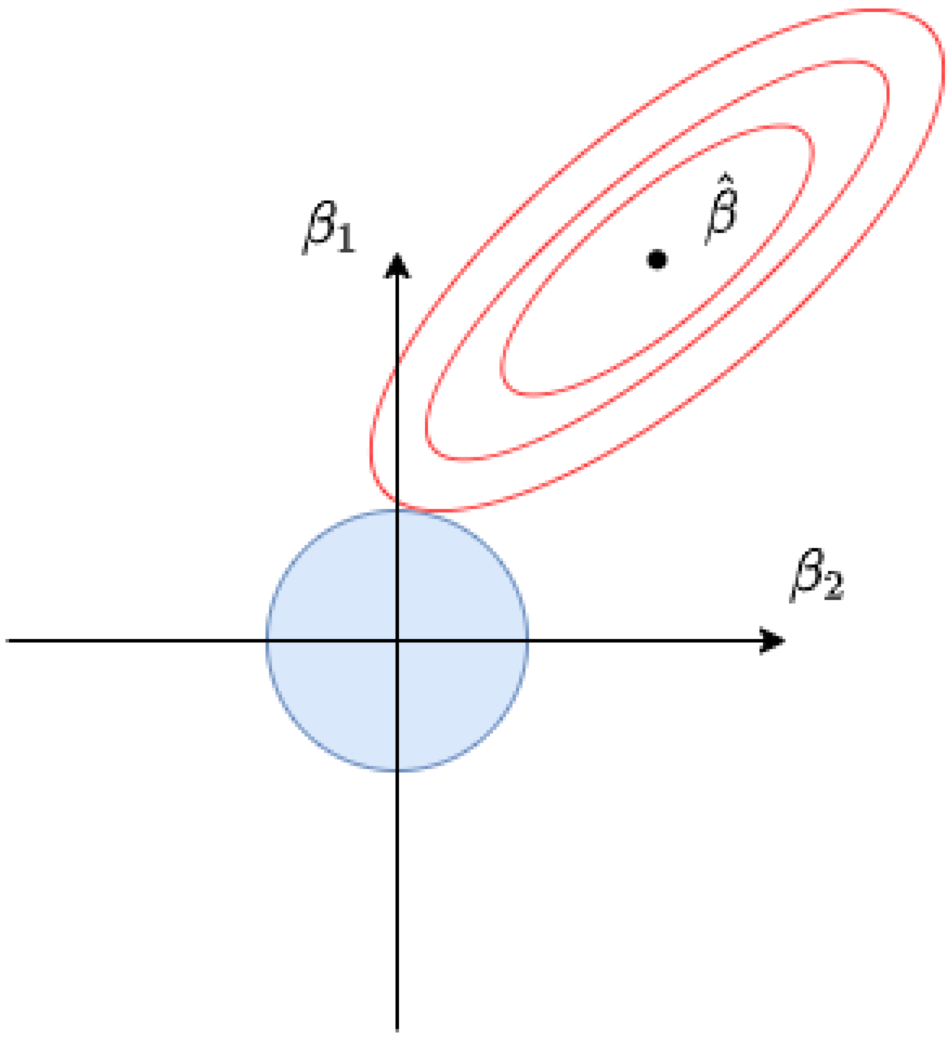


Figure 12: The constraints are given by $\beta_1^2 + \beta_2^2 < t$.

Understanding of the ridge regression will facilitate the understanding of the lasso regression, described in section 2.17.

2.17 Lasso regression

The *lasso regression* is another type of linear shrinkage method which also aims to penalize the coefficients of the simple linear regression model. The name is an acronym for Least absolute shrinkage and selection operator. It's similar to the ridge regression, but with a slight difference. The lasso estimate is defined according to equation (2.20) below or on Lagrangian form according to equation (2.21).

$$\begin{aligned}\hat{\beta}^{lasso} &= \arg\min_{\beta} \sum_{i=1}^{N}(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2. \\ &\text{subject to} \sum_{j=1}^{p} |\beta_j| \leq t.\end{aligned} \tag{2.20}$$

$$\hat{\beta}^{lasso} = \arg\min_{\beta} \Big\{ \frac{1}{2} \sum_{i=1}^{N}(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \Big\}. \tag{2.21}$$

The lasso estimates appears to be very similar to the ridge estimate, which is true, but there are some differences. Partly the factor in the beginning of the expression, but more importantly is the L_2 penalizing term β_j^2 used in the ridge regression replaced by the L_1 penalizing term $|\beta_j|$. This change of penalty term makes the solutions non-linear in y_i.

Setting t in equation (2.20) sufficiently small will lead to some of the regression coefficients to be equal to zero according to Hastie et al. (2001), which means that the lasso regression does a sort of continuous selection of the variables to be used in a model.

If t is chosen larger than $t_0 = \sum_1^p |\hat{\beta}_j|$, the lasso estimates are equal to the least squares estimates $\hat{\beta}_j = \hat{\beta}_j$. But t_0 can be chosen differently. If it is chosen as $t = t_0/2$, then the least square estimates are shrunk with about 50 % on average according to Hastie et al. (2001). Using lasso regression, some parameters can be shrunk all the way down to 0, in contrast to ridge regression. The lasso constraints for the two-parameter case is depicted in figure 13.

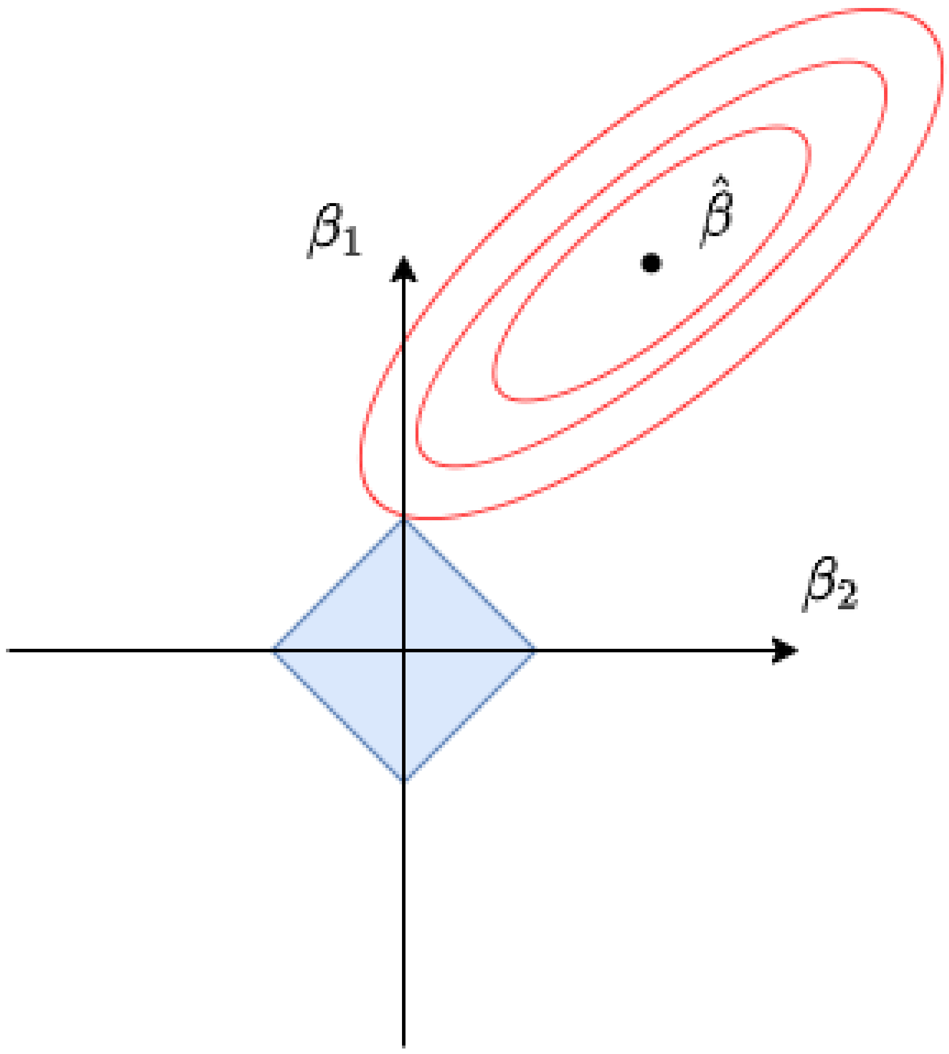


Figure 13: The lasso regression constraints. The constraints are given by $|\beta_1| + |\beta_2| < t$.

As seen in figure 13, the constraints marked in blue now takes the shape of a square rather than a circle as the ridge regression in figure 12. This slight change in penalty term means that the lasso regression allows some parameters, say $\hat{\beta}_j$ to be reduced all the way down to $\hat{\beta}_j = 0$, as stated by Hastie et al. (2001).

2.18 Hyper-parameters

When building a machine learning model of any kind, one thing that always needs to be finely tuned are the *hyper-parameters*. These are the parameters or 'settings' for the model. These settings often include such information as how many times the algorithm should train, what's the learning rate i.e. at what rate should the weights in the algorithm be updated and which optimizer to use. Understanding how these hyper-parameters affects the model structure and performance is often vital to achieve good results. However, since there are some many combinations of different hyper-parameters the process are often done through trial-and-error.

2.19 Outlier removal

When dealing with data for machine learning, it's often beneficial to analyze the outliers. One way to approach outliers is through the *1.5xIQR-rule*. This is also often displayed in boxplots. Firstly the interquartile range is defined according to $IQR = Q_{75} - Q_{25}$, where Q_{75} is the 75%-percentile and Q_{25} is the 25%-percentile. The 1.5xIQR-rule then states the following limits

$$\begin{aligned} upper\ limit &= Q_{75} + 1.5IQR \\ lower\ limit &= Q_{25} - 1.5IQR. \end{aligned} \tag{2.22}$$

Any datapoint falling outside of these limits can be suspected outliers. These limits are often depicted as lines within a boxplot. One common approach to outliers is to remove all of the datapoints falling outside of the interval in equation (2.22) according to Vinutha et al. (2018).

2.20 Standardization and z-score

Another important preprocessing aspect when working with ML is *standardization*. One popular way to transform data is through by using z-score. This changes the values of the data points to become zero mean, with unit variance. This is done by performing the transformation in equation (2.23) below,

$$z = \frac{x - \hat{x}}{\sigma}, \tag{2.23}$$

where x is the data value, $\hat{x}$ is the mean and σ is the standard deviation as stated byKaptein and van den Heuvel (2022). Edén Ohlsson (2021) also established that transforming the data is often useful when dealing with machine learning algorithms. According to Gal and Rubinfeld (2018) data standardization can lead to better machine learning and is especially useful when outliers has been handled according to Muhammad Ali and Faraj (2014).

2.21 Performance measurements

When working with different models, it is absolutely vital to be able to measure the performance of each model. Therefore different metrics is used to evaluate how well the models described the relationships in the data. The metrics need both to be able evaluate the models fit as well as the models generalization performance.

2.21.1 Root Mean Squared Error

The *mean squared error* is one measure which can be used when determining whether or not a model is fitted well to the data. The MSE is defined according to equation (2.24) (and also mentioned in equation (2.5)) below,

$$MSE = \frac{1}{N}\sum_{n=1}^{N}(y_i - d_i)^2, \tag{2.24}$$

where N is the number of inputs, the y_i is the true value for input i and the d_i is the predicted value for input i according to Sammut and Webb (2010). This is a tool often used when evaluating models and often goes by the name of squared residuals as well.

However because of the squaring in equation (2.24) the residual for input i does not have the same unit as the original target value y_i, which is why it is common to take the square root of the MSE to receive a measure with the same unit as the targets. The Root Mean Squared Error (RMSE) is defined by equation (2.25), with the same notations as previously stated above and in accordance with Chai and Draxler (2014).

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - d_i)^2} \tag{2.25}$$

This is a useful tool when comparing different models to each other, with the benefit of having the same units as the targets.

2.21.2 Mean Absolute Error

An alternate measure which also can be used when evaluating models is the Mean Absolute Error (MAE) in short. This is defined by equation (2.26) below,

$$MAE = \frac{1}{n} \sum_{n} |y_i - d_i| \tag{2.26}$$

where n is the number of inputs, the y_i is the true value for input i and the d_i is the predicted value for input i. The MAE will always be lower than the RMSE according to Chai and Draxler (2014).

One thing to note is that even though MAE provides additional information regarding the errors of the model, it performs better for uniform distributions. Since model errors are likely to have a normal distribution RMSE is often the better metric to consider when evaluating models as stated by Chai and Draxler (2014).

3 Methodology

In this section, the methodology of this thesis is described. It will explain the entire working-process so that it can be replicated and similar results produced. It will describe the different programs and tools used and it will also describe the preprocessing procedure as well as the two test used to evaluate the real-life applications of the models.

3.1 Gaining knowledge in the subject

To be able to grasp the concepts, difficulties and possibilities connected to measuring of CQ, a literature study were conducted. This literature study focused on different software metrics often used when analyzing source code as well as a deep dive in the concept of code smells.

A literature study on useful ML-algorithms was also conducted, to gain knowledge regarding several possible ways to build the model. Which algorithms would be preferred to use for these types of problems and which is not. The knowledge gained from these studies is displayed in the theory section, section 2, of the thesis.

3.2 Finding the tools

When the literature studies were conducted, the focus now shifted to the practical part of the project. The first step in the modeling were to set up a large, usable and labeled data set for the models to be trained on. This soon proved to be difficult using the data set provided by Ingka. This was because the data set only consisted of a small set of samples which would not be sufficient. There were some ideas regarding how to work around this problem, e.g. modifying the available data set at Ingka, either manually or by using data augmentation. However, after having dialogues with both supervisors, it was decided that the best way forward was to try to find an external data set with labels that contained different software metrics.

After spending some time looking for a usable, properly labeled data set, the one which was deemed the most suitable was the *Qscored* data set. The data set is described in more detail in section 2.1. It was decided that this data set contained the needed metrics to build a suitable model. The problem with the Qscored data set was that it solely consisted of metrics from programs written in Java and C#. The data set was constructed using tools only applicable to these two programming languages. In it self, this isn't a problem but since the software development team at Ingka writes its code in JavaScript (JS) in TypeScript (TS), there needed to be someway to acquire the same features to be used, but in the programming languages used at Ingka. A ER-diagram of the Qscored database can be found in appendix A.

After some investigating, the analyzing tool deemed the most appropriate was *SonarQube*. SonarQube is a code analyzing tool that is able to scan projects and extract metrics from many different languages. This includes the languages JS and TS which are the ones that are mainly going to be analyzed at Ingka. It can detect different kinds of code smells and also measure more common software metrics e.g. number of methods or code duplication.

The entire workflow of the developing models and making predictions on new data using Qscored and SonarQube is depicted in figure 14.

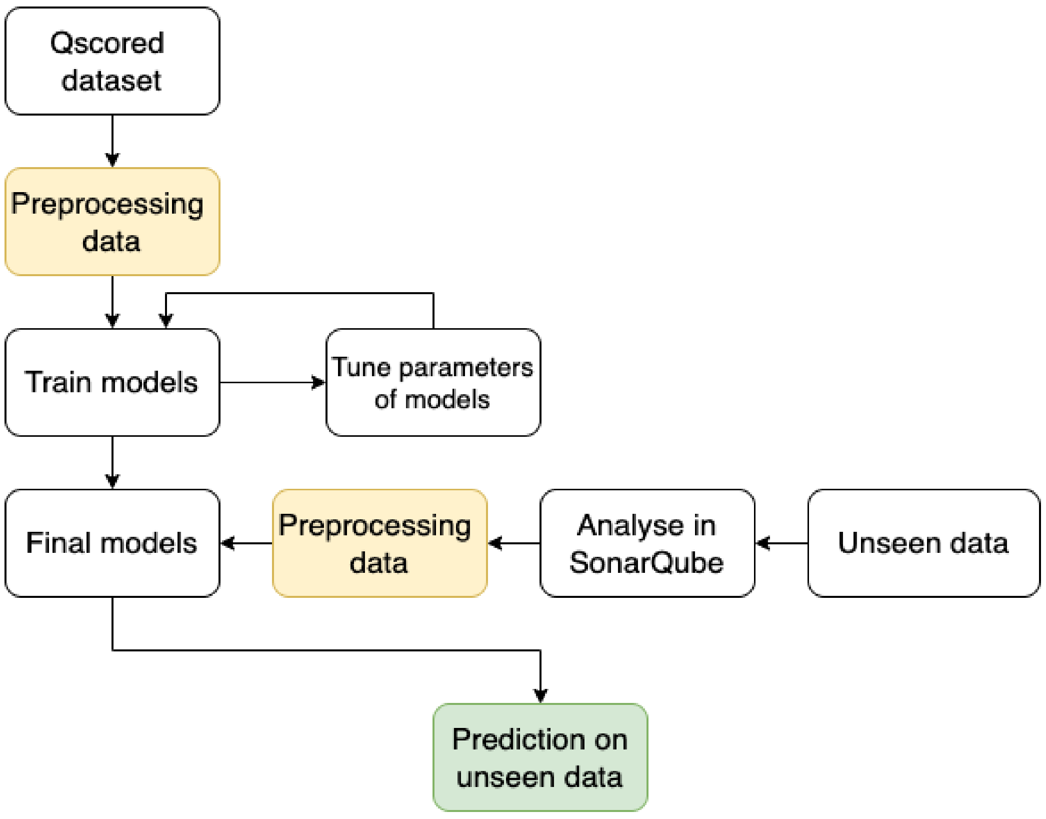


Figure 14: The process of developing the models and making predictions on new data.

3.3 Choice of metrics

Since the models were to be used when analysing code in JavaScript and Typescript, the metrics needed to be available in both the training data as well as the external analysis tool SonarQube. Only metrics available both in Qscored and SonarQube could be used, which were one basis on feature selection.

Using Qscored to train the data and using SonarQube when analysing new projects posed another problem. Metrics from a project analyzed in Qscored needed to produce roughly the same results in SonarQube. Therefore a small test were constructed where four files that were present in the Qscored data set were analyzed by SonarQube, to investigate the translatability between the two tools. This was another basis of which features to use in the models.

After examining the results and investigating if the metrics were roughly the same both from Qscored and SonarQube, the metrics which were deemed usable and translatable are depicted in table 2. CC represents cyclomatic complexity, described in section 2.5.5.

Table 2: The used metrics in the model.

Target variable	Metric 1	Metric 2	Metric 3	Metric 4	Metric 5
score	LOC	method_count	CC	code_duplication	total_smells

A correlation heat map of these metrics is shown in figure 15 below.

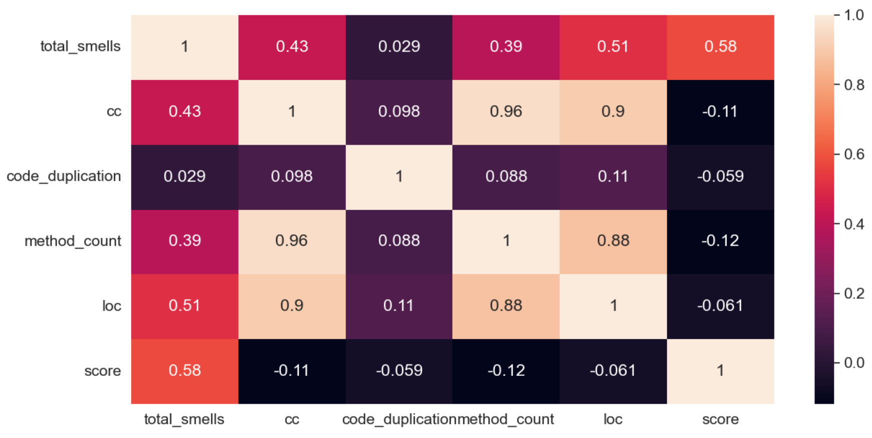


Figure 15: A correlation heat map of the five metrics used in the models.

As can be seen in figure 15 above, there is a strong positive correlation between the "score" and "total_smells". There are weaker negative correlations between "score" and "CC" and "score" and "method_count". Between the last two variables, "code_duplication" and "LOC" and the target variable there is a small negative correlation. There were some consideration to remove these variables, but since there might still be some correlation between that variables they stayed in the model. A simple correlation matrix only examines linear relationships between variables and there are possibilities for non-linear relationships between variables and the target variables.

3.4 Differences between Qscored and SonarQube

There is a discrepancy between the data base Qscored and the chosen analyzing tool SonarQube. The metric code smells is defined a bit differently with Qscored with respect to SonarQube. For example, SonarQube doesn't analyze so called *architectural* code smells, while they appear in Qscored. This might cause a problem, but was decided to be kept in the database when training the models. All types of smells were simply added together and the idea was to examine the metrics and see if Qscored and SonarQube returned roughly the code smell count.

SonarQube is also able to analyze projects in several more languages than Qscored. So if a project in the data base is analyzed using SonarQube, the analysis might consist of more files than the data base were able to investigate. If a project contains mainly code written in C#, but also containing some files in other programming languages such as python or HTML, the analysis of these languages will appear in SonarQube, but will not be included in Qscored.

3.5 Gather data points from the database.

For building a data set that included all the metrics that we wanted use from section 3.3, some tables in figure 23 in appendix A needed to be combined in order to be able to train a model.

The data was then filtered so it only included each project in Qscored once. Since one project could have been analysed at multiple occasions, only the latest analyse of each project was included in the data set.

First of all, the table solution contains most of the metrics we're looking for. It is only *cyclomatic complexity* that is not so trivial to retrieve. The code to get cyclomatic complexity in the correct way is in appendix C. After CC were extracted, the tables created in the data bases could be used to create the models.

3.6 Inspecting and dividing the data

In the Qscored data set only the files programmed in C# did contain the metric code duplication. The Java files did not since the evaluation tool which created the data set didn't have any way to measure code duplication for Java files. After some discussion with the supervisor at Ingka, a decision was taken to discard Java files since they lacked the metric code duplication. The reasoning behind this decision is that code duplication could play a role when measuring code quality and since it was available in SonarQube it was to be included in the model. The Java files were discarded and this reduced the number of samples from 86652 to 31337. This was considered to still be a large enough data set to construct a ML-model from.

Then the data was split in two separate parts. This was done to train one model on small projects with less than 5000 lines of code and one model on large projects with more than 5000 lines of code. The distribution of the entire data set is shown in figure 16, while the distribution after the divide is shown in figure 17a and figure 17b respectively. The data sets will from here on be out named as data set A for the data set with LOC < 5000 and data set B for the data set with LOC >= 5000.

The reasoning behind this split is as follows. The idea was to derive one set of models trained to be able to give predictions on smaller singular files. The other set of models were to give better predictions on larger projects, often consisting of more than 100 singular files.

The files to be analyzed and Ingka hardly ever exceeds a size of 5000 LOC, while the projects often are larger. The limit was drawn at 5000 to ensure that all smaller, singular files were to be predicted by the models trained at the smaller data set. The limit is arbitrarily chosen and could have been around 2500-3000 LOC as well.

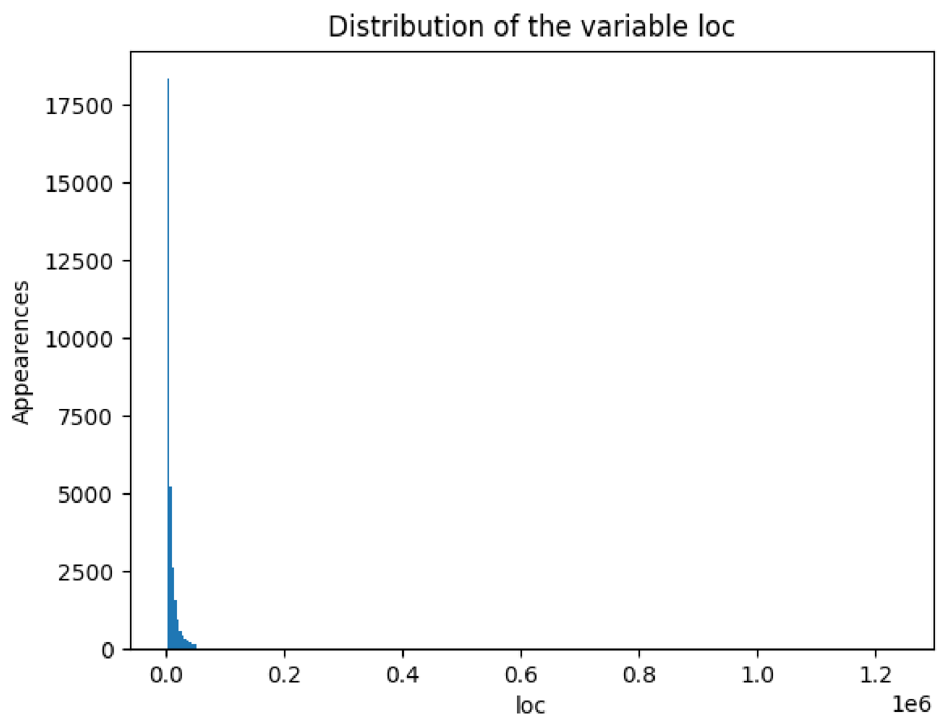
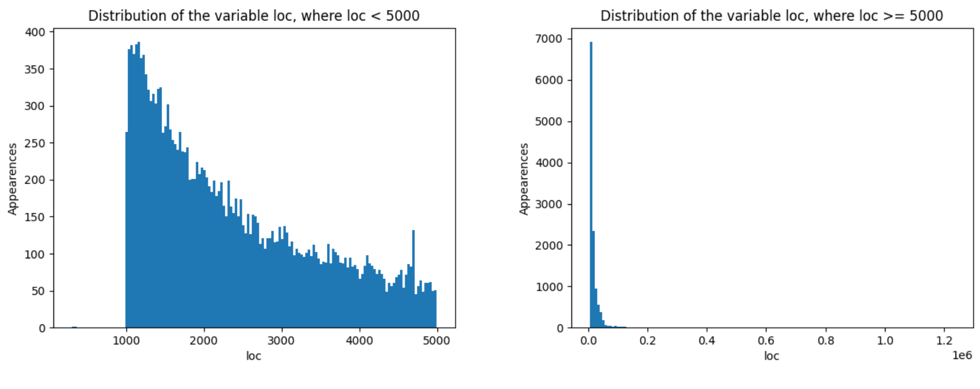


Figure 16: The distribution of the variable LOC for all data points.



(a)

(b)

Figure 17: The distributions of the variable LOC after the divide was made. On the left (a) the distribution for data set A and on the right (b) the distribution for data set B.

The specifics for each of the two data sets are shown in table 3 below.

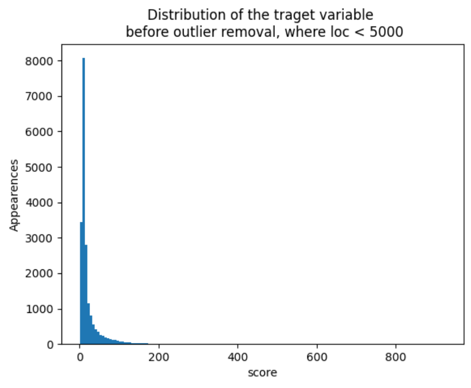
Table 3: Specifics for both of the data sets.

LOC	Min	Max	Mean	Number of data points
< 5 000	300	4 999	2366.13	19 575
>= 5 000	5 000	1 236 196	19 867.61	11 762

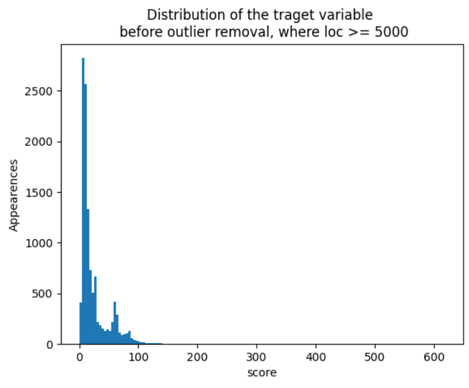
Table 3 shows more than figure 17, especially for figure 17b, since there are some data points which really distorts the distribution. The largest data point has a LOC of 1236196, but the mean for data set B is rather 19867.61, which gives a more true insight about the variable LOC in the two data sets.

3.7 Preprocessing data

Before the models could be trained, the data needed to be preprocessed. First the data was inspected with regards to boxplots. The target variable was especially considered since it had a lot of outliers. The distributions and boxplots for the target variable score was created for both data sets and are depicted in figure 18 and figure 19.

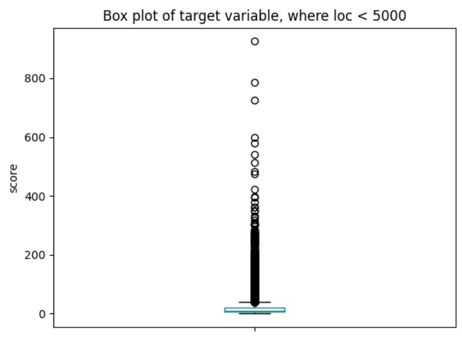


(a)

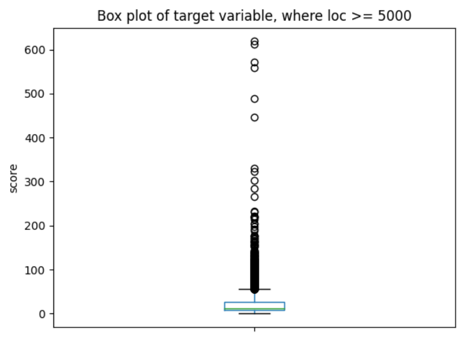


(b)

Figure 18: The distribution of the target variable(score) before outlier removal, on the left (a) for data set A and on the right (b) for data set B.



(a)



(b)

Figure 19: Two boxplots of the target variable score, on the left (a) for data set A and on the right (b) for data set B.

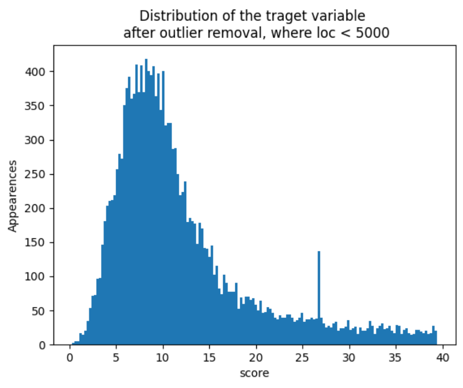
The outliers were removed with regards towards the target variable. This was done using the 1.5xIQR-rule described in section 2.19. The IQR statistics for both data sets can be seen in table 4.

Table 4: The IQR statistics for both data set A and B.

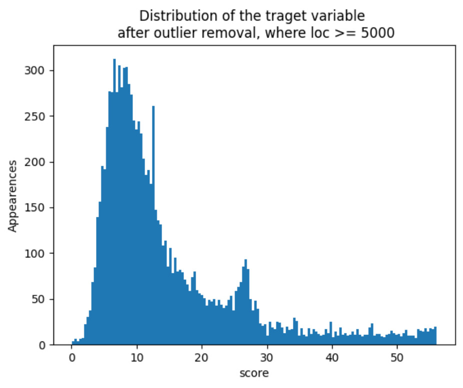
data set	25%-quantile	75%-quantile	IQR	Lower limit	Upper limit	Number of outliers
A	7.42	20.24	12.82	-11.81	39.47	2 580
B	7.91	27.20	19.29	-21.03	56.14	1 651

No outliers below the box were removed, since there are no negative values for the target variable. Each data point with a score greater than 39.47 were removed for data set A and each data point with a score greater than 56.14 were removed for data set B. The data points considered are the ones outside the boxes in figure 19 and the removal of these reduced the number of data points with 2580 and 1561 data point for data set A and B respectively.

The distributions after the removal of the outliers in the target variable is shown in figure 20 below.



(a)



(b)

Figure 20: The distribution of the target variable(score) after outlier removal, on the left (a) for data set A and on the right (b) for data set B.

After the outliers were removed, all data was standardized in accordance to the z-score transformation described in section 2.20. The target variable score was kept untransformed. Note that any predictions on new data need to be transformed with the same standardizer which was used to scale the training data.

When the data had been preprocessed, another split could be made on the data sets, to create a training set and one test set. The models were, as the name suggests, trained on the training sets and tested on the test sets. The data points were randomly chosen and the percentage between training and test were 80% to 20%.

3.8 Building the models

After the data had been preprocessed and split into training and test sets, the models could start to be built and optimized. The procedure of finding good models was done on both data sets. This was done since there was an idea of differentiate models. Some models were trained on large projects containing hundreds of files and some focused on individual, singular files.

The building of the models was done using Python and more specifically by using the TensorFlow package by Abadi et al. (2015) and the Scikit-learn package by Pedregosa et al. (2011). The initial testing used were performed using the default hyperparameters as seen in each respective python package. The built-in packages provide simple and efficient tools when applying many types of different machine learning algorithms.

Initial testing started with seven different types of models, which are presented in table 5 below.

Table 5: The initially tested algorithms.

Model	Algorithm used
1	Linear regression
2	Decision tree regression
3	ANN
4	Random forest
5	Gradient boosting
6	Lasso regression
7	Ridge regression

This selection of algorithms were deemed to represent three different types of algorithms. Linear, lasso and ridge represent the linear regression family, the decision tree, random forest and gradient boosting represent the decision tree family while the ANN represent the neural network algorithm.

The artificial neural network was built and was tested with a lot of different nodes and number of layers until we settled on this final model. The process to find the optimal architecture and hyperparameters of the nerural network is always a difficult process. There was a lot of trial and error to find an architecture which produced good results.

The hyperparameters for the different regression models as well as the decision tree-algorithm were tuned by testing different learning rates, depth and number of estimators.

3.9 Comparing the models

The models were tested on the testing data and the different models were compared with each respective RMSE. The MAE was briefly taken into consideration, but the measure which was the deciding one were the RMSE. The two models with the lowest RMSEs were deemed the most suitable when evaluating real life applications of the models.

3.10 Gather software metrics from projects

As mentioned before, SonarQube was the tool used for analyzing new files and projects and to gather software metrics. The process was pretty straight forward, making api calls to get files and then get metrics from the files. These metrics were than transformed with the same scalar used when transforming the training data.

3.11 Evaluating the model

The RMSE can be used to compare different models, but it was also desirable to validate the model on real life data, rather than just the models performance on the Qscored dataset. Therefore two tests were created, which would evaluate two possible applications of the models. Firstly the possibility to compare different singular files with one another and secondly the possibility to evaluate the same project over time to examine if the code quality is increased over time. The tests are described below in section 3.11.1 and 3.11.2 respectively.

3.11.1 Test 1

To test the models in real life appliances, *test 1* was designed. The test consisted of letting the two models make predictions on internal Ingka files from both front end and back end. Four front end files and four back end files were chosen. After the models had evaluated the files, three developers from Ingka with varying working experience ranked the files individually in terms of code quality. The working experience of each developer is presented in table 6 below.

Table 6: Experience of the three developers, both at Ingka and in software development in general.

Developer	Time at Ingka (years)	Time working in software development (years)
1	1	5
2	1	20
3	3	21

The form sent out to developers with their instructions can be found in appendix D.

The files were partly chosen based on the score the models predicted. The idea was to choose files with varying evaluations by the models, to give a selection of different scores. All of the developers gave feedback for each file about the code quality and then ranked the files according to code quality. When the results from the developers were received, it was compared to the predictions made by model.

3.11.2 Test 2

In addition to the opinions of the developers, *test 2* was created. This was done by evaluating one project at Ingka at different points in time. By using the history of commits to GitHub, it was possible to let the models evaluate the same project and see if the quality of the code was improved. This is also thought to be a possible application of the model.

The models tested the project ten times during the span of ten months. The first data point was simply a copy of the old version of the project, which was later reduced and refactored, which is why the project initially is very large. The assumption after discussing with the developers responsible for the project was that the the code quality were to be improved over time and the score predicted by the models were therefore to decrease over time.

The reasoning behind the assumption that the code quality would increase over time was due to the opinion of the developers. The developers pointed specifically to the following problems.

- Not a uniform terminology.
- Not a uniform structuring of the logic.

- Difficulties to keep developing the program.
- Difficulties to maintain.

These were the aspects of the code which were improved by the developers during these months and the reason to why the CQ should increase. This would result in a decrease in the score predicted by the models.

A decision was made to exclude Ingkas internal test files since they increase the complexity but won't be used for the real application. This was taken after consultation with the responsible developers, which declared that the inclusion of these files would not be representing the quality of the code. These files increases both the complexity and size of the project. The files doesn't affect the functionality of the program and it would therefore be misleading, according to the developers, to include these files when discussing the projects CQ. Also they were not in the first file so it would not be difficult to compare the project with or without test files.

This removal is done by excluding all files that had the string `.test.` or `cypress/` in the file name. The data was then saved in a dataframe with software metrics and the file path. To find the metrics for the entire project, all the metrics were summed from each individual file to get the total value.

4 Results

In the following section, the results will be presented. Firstly the results from the two carried out tests will be presented. Secondly the RMSEs and MAEs will be presented for both the inital testing of the models and for the final models used in the tests. Lastly four analyses made by SonarQube and Qscored will be presented and compared.

4.1 Evaluation of model

4.1.1 Results, test 1

From the form sent to the developers the following results were obtained compared to the rank of our model. For the front end files, table 7 displays the rank given by the developers and the two models. "1st" is the file with the highest code quality and "4th" is the file with the lowest.

Table 7: Rank of front end files according to developers and the model

rank	Developer 1	Developer 2	Developer 3	ANN Model	RDF Model
1st	file 3	file 3	file 3	file 3	file 3 and file 4
2nd	file 4	file 4	file 4	file 4	-
3rd	file 2	file 1	file 2	file 1	file 1
4th	file 1	file 2	file 1	file 2	file 2

They also gave their reasoning for the given rank.

Developer 1: I think files which is easy to read usually is easier to maintain, has lower complexity, higher robustness and easier to catch errors. It was a close call between file 3 and 4, but file 3 is easier to maintain with less duplicated code even though the file has higher complexity with nested switch statements.

In the bottom of the list, it was close between file 1 and 2. In the end I decided to rank file 2 above 1 because it had less commented out code and less comments which I did not know what they meant. I also think it would be easier to fix the issues in file 2.

Developer 2: Given that we are a development team the files 1 and 2 are very hard to add logic too and in addition very error prone since too many functionalities.

Developer 3: The rank is motived having in mind your input related to the complexity, the maintainability, the robustness and error handling of the code.

The model gave predictions displayed in table 8 below.

Table 8: Score of the files used for the front end evaluation.

File	ANN Score	RDF Score
1	4.7543	2.03
2	4.9509	2.32
3	3.9674	1.5956
4	3.9696	1.5956

The ranking of the back end files are presented in table 9 below.

Table 9: Rank of back end files according to developers and the model.

rank	Developer 1	Developer 2	Developer 3	ANN model	RDF model
1st	file 4	file 2	file 2	file 4	file 2
2nd	file 2	file 4	file 1	file 1	file 3
3rd	file 1	file 1	file 4	file 2	file 4
4th	file 3	file 3	file 3	file 3	file 1

Below is a transcript of the developers reasoning regarding the back end files.

Developer 1: File 4 is very easy to follow because of low complexity and high readability. Thus, being maintainable and less risk for errors slipping by.

File 2 is also easy to follow with good documentation of public methods. File 3 is easy to follow.

It was a close call between file 2,3 and 4, but in the end the ordering was decided by the time I had to spend to understand what the code was doing. File 3 was the least understandable with higher complexity. Being a large file and function it took longer time to understand and evaluate function, robustness etc.

Developer 2: I like class based files, Also I think the createUser file does too much. There is no structured way of doing it.

Developer 3: The rank is motived having in mind your input related to the complexity, the maintainability, the robustness and error handling of the code.

The model gave the following score:

Table 10: Score of the files used for the back end evaluation.

File	ANN Score	RDF Score
1	3.9397	1.608
2	3.9602	1.474
3	3.9787	1.509
4	3.7436	1.520

4.1.2 Results, test 2

The next test was to evaluate a project at different points in the developing process. The evaluation was done using the neural network as well as the RDF trained on data set B. The metrics results are presented in table 11 and presented as a time series in figure 21.

Table 11: Dates, metrics and scores for the same project evaluated at different points in time. Evaluated both by ANN- and random forest-models.

Version	Date	LOC	method count	CC	CD %	Smells	ANN score	RDF score
1	21-06-14	53 346	1560	3011	21.0	413	1.4647	2.5174
2	21-08-10	67 894	1400	2627	61.8	437	1.3437	2.445
3	21-09-15	29 408	595	1013	13.2	132	1.3188	1.8882
4	21-10-12	29 894	605	1033	13.0	142	1.3265	1.8697
5	21-11-30	31 259	698	1188	11.9	161	1.3362	1.8755
6	21-12-23	32 965	763	1356	11.2	156	1.2939	1.8461
7	22-01-31	33 258	814	1460	5.1	142	1.2466	1.7461
8	22-02-21	33 648	847	1542	5.0	131	1.2215	1.7649
9	22-03-14	33 819	915	1673	5.3	131	1.2244	1.7813
10	22-04-14	34 425	971	1765	4.8	123	1.20114	1.7439

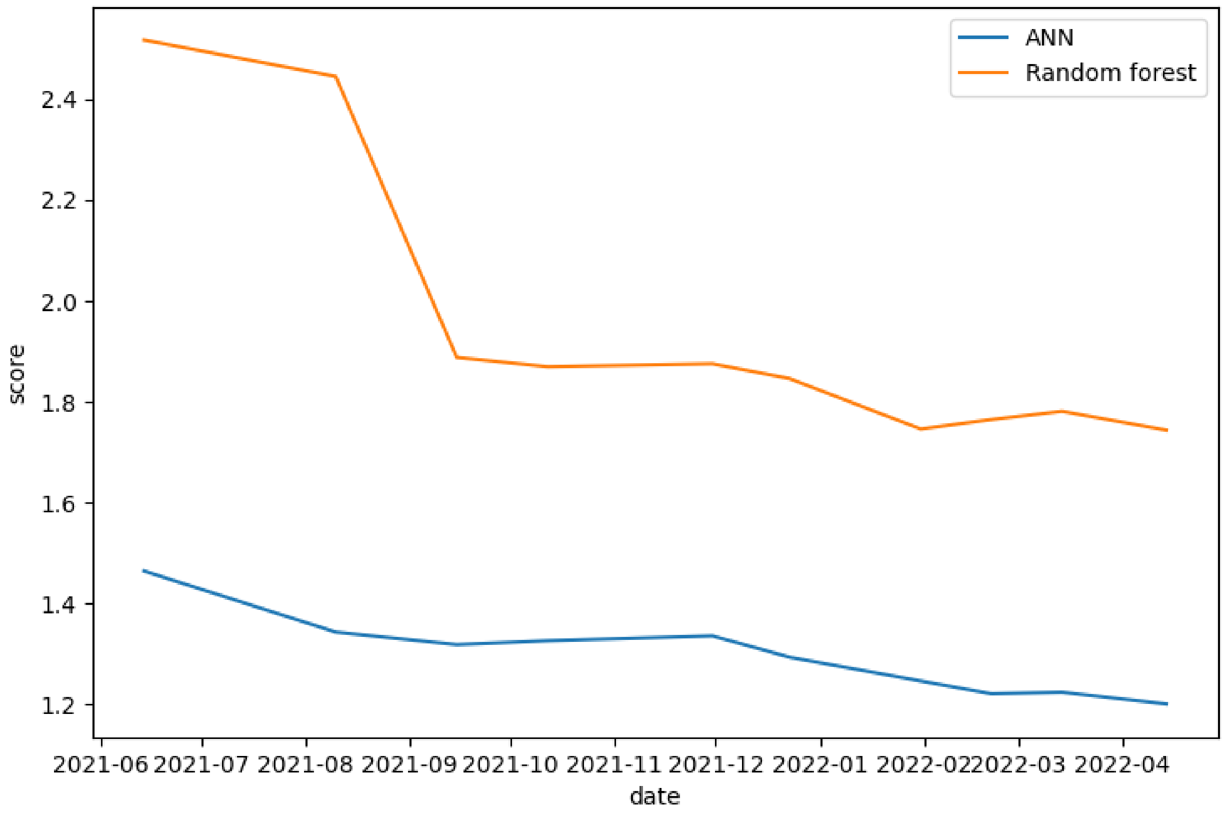


Figure 21: The evaluation over the same project over time, at ten different dates during the span of ten months. The evaluation is made both by the ANN (blue) and the random forest (orange).

4.2 RMSEs, MAEs and hyperparameters for respective different models

There was some initial testing done, without tuning of any hyper-parameters with models that were deemed useful for the problem. The initial training results for both data sets are presented in table 13, with both the RMSE and the MAE.

The architecture and hyperparameters of the initial ANN is shown in table 12.

Table 12: The architecture and hyperparameters of the initial ANN.

Layer	**Type**	**Number of nodes**	**Probability**
Input	Dense	5	-
Hidden 1	Dense	100	-
Hidden 2	Dense	100	-
Hidden 3	Dropout	-	0.01
Hidden 4	Dense	100	-
Output	Dense	1	-
Hyper-parameters			
Learning rate, η	0.001	**Epochs**	50
Weight initalizer	random	**Activation functions**	relu
Optimizer	rmsprop		

Table 13: The initial seven algorithms RMSEs and MAEs from the test set for both data sets.

		data set A		data set B	
Model Index	Algorithm	RMSE	MAE	RMSE	MAE
1	Linear regression	3.40149	2.34368	9.32053	6.22647
2	Decision tree	1.43261	1.00895	1.69959	1.07359
3	ANN	1.14416	0.79933	1.27581	1.01420
4	Random forest	1.01560	0.72248	1.22203	0.71701
5	Gradient boosting	1.10727	0.80212	1.53488	1.04683
6	Lasso	3.95056	2.70239	9.53730	6.95084
7	Ridge	3.40145	2.34356	9.32107	6.22706

According to the initial testing, the models which performed the best were the different types of decision trees as well as the ANN. The linear regressors were performing rather poorly. This was true for both data sets, even though every algorithm performed slightly worse for data set B.

Following of these initial results, attempts to improve the models were done by tuning the models different hyperparameters. However, most of the models were performing best with the initial hyperparameters. The algorithm which was improved by fine tuning was the neural network. This was done by changing the architecture of the network as well as trying out different activation functions and learning rates.

The final architecture of the neural network consisted of one input layer, seven hidden layers and one output layer. The learning rate were set to $\eta = 0.001$ and it was trained over 100 epochs. This resulted in a final RMSE of 1.11961. However since there are infinitely many way to construct a ANN there might be other structures which gives a lower RMSE. The RMSE = 1.11961 was however deemed reasonable and used in the evaluation of Ingkas internal projects.

The architecture of the ANN is displayed in table 14 and the loss during 100 epochs is depicted in figure 22 below.

Table 14: The architecture and hyperparameters of the final ANN.

Layer	**Type**	**Number of nodes**	**Probability**
Input	Dense	5	-
Hidden 1	Dense	100	-
Hidden 2	Dense	100	-
Hidden 4	Dropout	-	0.01
Hidden 5	Dense	100	-
Hidden 6	Dense	100	-
Hidden 7	Dense	100	-
Hidden 8	Dense	100	-
Output	Dense	1	-
Hyperparameters			
Learning rate η	0.001	**Epochs**	100
Weight initializer	random	**Activation functions**	relu
Optimizer	rmsprop		

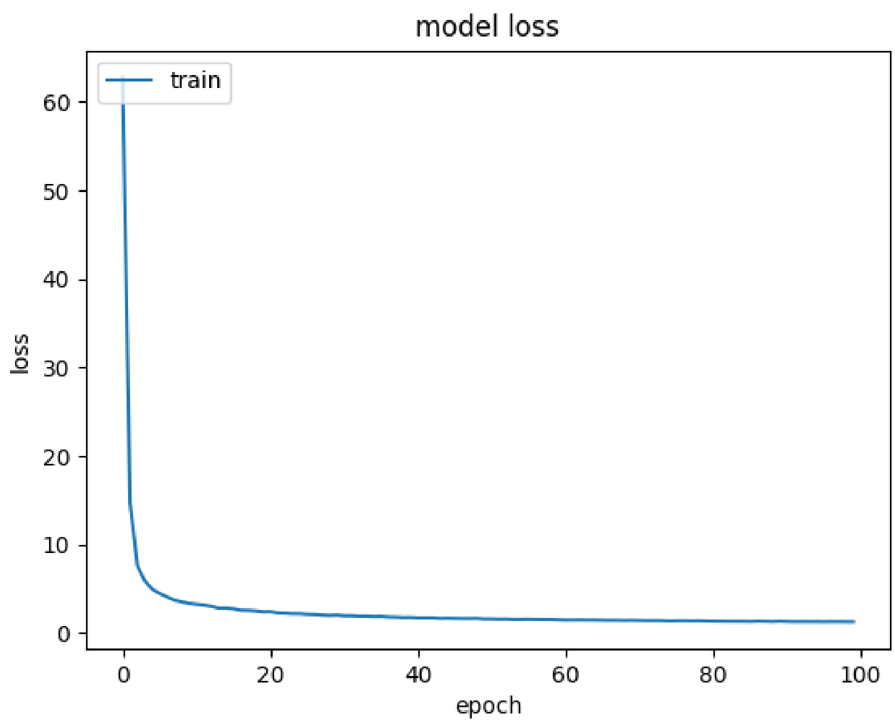


Figure 22: The loss over 100 epochs for the final neural network.

In table 15, the final RMSEs and MAEs of every algorithm is shown for both data sets. These result were the reasons that the random forest and neural network were the two models used when evaluating real life appliances. Note that it's only the values for the ANN which have changed from table 13.

Table 15: The initial seven algorithms RMSEs and MAEs from the test set for both data sets.

		data set A		data set B	
Model Index	Algorithm	RMSE	MAE	RMSE	MAE
1	Linear regression	3.40149	2.34368	9.32053	6.22647
2	Decision tree	1.43261	1.00895	1.69959	1.07359
3	ANN	1.11961	0.75398	1.27581	1.01420
4	Random forest	1.01560	0.72248	1.22203	0.71701
5	Gradient boosting	1.10727	0.80212	1.53488	1.04683
6	Lasso	3.95056	2.70239	9.53730	6.95084
7	Ridge	3.40145	2.34356	9.32107	6.22706

4.3 Comparing metrics

It was important that the metrics were valued same, or at least in the same magnitude, both in the external data set that the model were trained on and the metrics received from SonarQube. Because of this, SonarQube were tested on projects from the external dataset and the metrics compared. The metrics from four open-source projects analyzed both in SonarQube and Qscored are presented in table 16 to table 19.

Table 16: Project 1: ecomms.

Metric	SonarQube	Qscored
LOC	1221	1551
Number of methods	104	110
CC	206	126
code duplication	1.9%	1.61 %
Smell count	55	41

Table 17: Project 2: glazedlists-tutorial. Note that this is a Java-file, hence why the metrics CD is missing.

Metric	SonarQube	Qscored
LOC	1292	1292
Number of methods	90	97
CC	175	175
code duplication	38.6%	-
Smell count	65	236

Table 18: Project 3: StjMbs.

Metric	SonarQube	Qscored
LOC	3349	2835
Number of methods	262	146
CC	332	179
code duplication	2.9%	0 %
Smell count	49	99

Table 19: Project 4: WeatherService.

Metric	SonarQube	Qscored
LOC	1598	1917
Number of methods	81	59
CC	140	102
code duplication	0%	0 %
Smell count	290	230

Since these files are open source, the reader can access these projects and the links will be located in appendix B.

5 Discussion

5.1 On the overall performance in the different tests

Inspecting the results from test 1, where the models was trained on data set A, they showed that the small models performed quite well compared to developer rankings and opinions. It performed especially well for the front end files as seen in table 7. Developer 1 and 3 slightly disagreed with the models at one point and ranked it differently. However the developers still agreed with the models in two out of four cases. This obviously gives the models some validity, that it overwhelmingly agrees with the developers in most cases for the front end files. Developer 1 pointed out that it was a close call between file 3 and 4 for being the best file and the same with file 1 and 2 for being the worst. It's interesting that the RDF model agrees to such a large extent with developer 1, that it gives the two files the exact same score as seen in table 8. The models agree with this and file 3 and 4 are closely matched in terms of score. This shows that the models at least can effectively draw a conclusion which are the worst files. This will make it easier for the company in the future choose which files that needs to be rewritten.

In the case of the back end files, there are bit more discrepancy between the models predictions and the developers. This is seen in table 9. The models agrees with all three developers regarding which file has the lowest CQ, but only agrees with one regarding which one is the best. It agrees with one developer regarding which file has the second highest CQ and not agreeing with one developer regarding the third best. This might be an indication that the models are more applicable to predict front end files rather than back end files. However, looking at table 10, the spread of scores is not as large for the back end files compared to the front end files. This might also be a reason to why the developers and the models didn't agree quite as clearly as for the front end files. There isn't large differences between the CQ of the files for the back end, according to the models.

Regarding test 2, where the models were trained on data set B, the score were expected to be lowered throughout the different months assuming the code were refactored and improved. This was the case as seen in table 11 and figure 21, which might indicate that the models might be used to evaluate a project over time to examine if the quality of the code is improved.

5.2 On the design and potential weaknesses of the tests

One problem with test 1, described in section 3.11.1, is that it is susceptible to bias from the developers. There's a lot of room for the developers own opinions. This could be problematic since what one developer considers bad code, another developer might approve of. However, looking at the experience of the developers seen in table 6 their requirements for what is considered to be high quality code is thought to be rather similar. It was also problematic to find back end files which gave a large spread in the scores. This might be an explanation to why the ranking differs to the ranking of the developers.

Regarding test 2, described in section 3.11.2, the test was conducted under the assumption that the code quality improved over time. The results in figure 21 agrees as well as the asked developers, but there's no guarantee that this is the case. There might still be that the code quality over time doesn't always decrease, even though the results indicates it, and the models predictions might be misleading.

5.3 On the performance of different models

As seen in table 15, the algorithms with the lowest RMSEs were the decision tree algorithms e.g. random forest and gradient boosting, as well as the ANN. All of the linear regressors performances were rather poor and produced high RMSEs compared to the other models. The ones which performed the best were the random forest, gradient boosting and ANN.

One aspect that the writers of the thesis found interesting were that the gradient boosting algorithm had slightly worse performance than the random forest algorithm. This was surprising, since the authors expected gradient boosting, which incorporate earlier errors in the model, to perform better than the random forest algorithm. This might be due to noise in the data which the gradient boosting accidentally models while the random forest doesn't. The gradient boosting often performs better than random forest, but might have problems with noisy data which is a possible explanation to the results.

The reason behind the poor performance of the linear regressors could be that the relationship between the target variable and the other variables is non linear. Then a linear regressor gives poor results which table 13 shows. Then for example a random forest will find other more complex relations and the results shows this is the case. Also the neural network will find non linear relations from the ReLU activation functions. If the relationship is non linear, it is possible to model it with the decision tree algorithms as well as the artificial neural network. However, both ridge and lasso is part of the linear regression family and it can therefore not overcome this obstacle.

Another explanation to why the ridge and lasso regressors performs poorly on the test set is due to the fact that they might be reducing variables which might actually have an impact on the models. There is always a risk when reducing variables that their shrinkage worsens the results. This could of course be the case in the neural network as well, but the low learning rate might counteract it.

5.4 On the external data set

One major problem with the thesis and the building of the models was to find a large, labeled data set. When Qscored was discovered it was a great step forward in the process, but it introduced another problem. If a model is to be trained on a external data set, how certain is it that the labels in Qscored are true? At some point, a ground truth must be established, but how is this decision taken? The Qscored data set were considered to be valid because of the interviews that was held during the creation of the data set. The opinion of 31 professional developers is deemed to validate the data set and make it a useful truth to optimize against when training our models. There is also a discussion about if the concept of "good" quality code changes over time, but since the data set was created in 2020 it's assumed to be up to date.

Another aspect of the Qscored data set is that it only built using Java and C#. These are two very common programming languages, but at the department at Ingka, JS and TS is almost exclusively used. This introduces a problem regarding if the model can be used on a programming language different from the one it was trained on. Java and C# code might have another structure to the other languages and might contain i.e. more or less logic than a program written in JS and TS. One idea might be to use the model, but at another department at Ingka which is using more Java or C#, but since there is a lack of labeled data for JS and TS this was deemed as an acceptable solution.

5.5 On the difference between Qscored and SonarQube

One difficulty when discussing modeling of code, or rather the exchangeability between different analysis tools, is the different definitions of metrics. The models are trained on metrics from Qscored, but when analysing the same projects in SonarQube, the metrics are not identical. This is seen in table 16 to table 19. These differences were especially apparent with regards to the code smell metric. This might of course lead to problems when making predictions on new data. The models would probably benefit from training on another data set with data points more similar to the ones it will make new predictions on.

When comparing the Qscored to SonarQube, there are some differences how the metrics are counted. However, since the metrics are rather similar for the projects, at least in the same magnitude, this was deemed acceptable.

5.6 Metric selection

As mentioned in section 3.3 the five metrics that were used in the data set was LOC, number of methods, CC, code duplication and number of smells. The fact that these metrics were available both in Qscored and in SonarQube was one reason to include these five features. The results in section 4.3 shows that the metrics for four different project compare relatively well and is often in the same magnitude. It should be noted that code smells are calculated a bit different between Qscored and Sonarqube. Qscored has more of a focus on architecture and design smells while SonarQube mostly focus on implementation smells. Then the question was whether we would include design and architecture smells in the data set for training our models. After comparing different combinations of implementation, architecture and design smells, it was chosen to use all three combined and since the magnitude is roughly the same, this was accepted.

Another topic when selecting metrics is how to select which ones to include in the models. What impact will they have on the models? As seen in figure 15 the most dominant variable is the "total_smells" variable. This however is expected since the original score was calculated using the smell density. The other variables have lower linear correlation with the lowest being the "code duplication" and "LOC". These might not have an impact on the models, at least not a linear one, but were still kept. There might still be some relationships between the two variables and the target variable, even tough they might be rather weak.

5.7 Addressing the research questions

The research questions stated at the start of the thesis were the following.

- **Q1.** Is it possible to construct a ML model to evaluate a company's software code quality?
- **Q2.** In the specified data set, which ML algorithms achieve the best performance?
- **Q3.** What are the possible usages of such a model?

These questions are addressed below.

Q1. The findings of this thesis points to the fact that there are possibilities to construct a ML-model to evaluate code quality. There might be better models than the ones used in the thesis, but the conclusion is that there at least are possibilities. The findings seem to point at the models performing better at front end files rather than back end files according to test 1.

The data set is at the moment the best one to be found for modeling code quality, but it is not perfect. If there were some software engineers who took the time and reviewed a large amount of files and labeled them, that would be a much better way to build a data set for training a machine learning algorithm. But the results still show that it is possible to evaluate code quality.

Q2. Using the Qscored data set, the models which achieved the best performance were the different types of decision tree-algorithms as well as the ANN. The linear, ridge and lasso, regressors were performing poorly for this specific problem. This was seen when testing the models on the external data set and comparing RMSEs. This might point to a non-linear behaviour between the input variables and the target variable.

Q3. The results from test 1 indicates that it is possible to use the models to compare singular files with one another as well. This is also a though application, to measure which files have the higher code quality and which files has the lower.

As figure 21 shows, it is possible to follow how the code quality develops over time. In the case shown in the results, the code quality increases over time which is the desired result. The models agrees with the assumption that the code quality is improved over time.

Regarding the decision to make to distinct models, one for larger aspects such as entire projects and one smaller to investigate CQ of singular files. This was thought to improve the models and make separate ones for large and small files. One idea could also be to make one model to increase the models ability to generalize, but a decision was taken to create two versions.

6 Conclusion

In conclusion, a model for estimating code quality was built and seemed to work reasonably well. This model works especially well for front end files where it is able to distinguish well written files from more poorly written files. For back end files, the same conclusion could not be drawn. The model is also able to get a measure for the code quality in an entire project consisting of a greater number of files. This could help a company to keep track whether the code quality of a project improves or decreases over time. Then the model can also be used to point out which files that needs to be improved the most to improve the code quality.

The findings of this thesis shows that it is possible to model code quality using machine learning-algorithms and it would be an area to further explore. The artificial neural network, random forest algorithm and gradient boosting all performs well, while the different linear regressors perform poorly. The findings also points to the fact that the models are able to evaluate and compare singular files, while also being able to evaluate larger projects over time, which are the two main usages of the models. The models appears to be able to rank front end files more successfully than back end files. The models are not planned to point to exact points in source code which need to be refactored, but rather to give an indication on which files might need to be restructured. There is a discrepancy between the data set used for training and analyzing tool which might cause problems when making predictions on new data. The process would probably improve if the models were trained on a data set created by analyzing projects in SonarQube.

Overall, the thesis is shedding new light on the incorporation between software development and machine learning. The two areas can be intertwined successfully and the writers of this thesis think that the possibilities of combining software development and ML are great. There might be a lack of labeled data in the software development field, but as long as there is large data bases to train models on, there exists a lot of benefits of combining ML and data science.

7 Future work

This thesis shows the potential for modeling code quality in terms of software metrics, the final model did not include a lot of metrics since there were only so many that matched between Qscored and SonarQube. One way forward could be to try to include more abstract variables in to the model, such as how many persons would have worked on the project, for how long etc. This would enhance the model and give it even more reach. That would make it more of a technical debt problem rather than just code quality. If this is done, the models could show which files and projects that are too dependant on one person for example.

Another future continuation would be to build a data set more fit for the purpose at hand. That would include scanning projects for software metrics in the same program and language as the evaluation is made. This would make the correlation between the data set and data which the model is used on higher. It would also be a good idea to for each file/project to be evaluated by experienced software developers have better labels for the data set. The problem is that it would be rather time consuming to evaluate so many files and that it would still be a subjective opinion from the developers, some developers would think differently about different files and what is important.

This could prove an important aspect for any software developing company. They could take time to periodically evaluate their own projects and files, to start building their own labeled data set. This could for example happen during one day every third month. Like it is with a lot of collections of data, it's often preferable to start earlier rather than later. Software developing companies could greatly benefit from having data on the quality of the written code.

References

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

D. R. M. Almuttairi. What's the difference between the front- end and back-end?

L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.

T. Chai and R. R. Draxler. Root mean square error (rmse) or mean absolute error (mae)? – arguments against avoiding rmse in the literature. *Geoscientific Model Development*, 7(3):1247–1250, 2014. doi: 10.5194/gmd-7-1247-2014. URL https://gmd.copernicus.org/articles/7/1247/2014/.

P. M. Edén Ohlsson. *Introduction to Artificial Neural Networks and Deep Learning*. Media-Tryck, 2021.

F. Fontana, M. Mäntylä, M. Zanoni, and A. Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21:1143–1191, 2015.

M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., USA, 1999. ISBN 0201485672.

R. S. Freedman. Testability of software components. *IEEE transactions on Software Engineering*, 17(6):553–564, 1991.

M. Gal and D. Rubinfeld. Data standardization. *SSRN Electronic Journal*, 01 2018. doi: 10.2139/ssrn.3326377.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016. ISBN 9780262035613. URL https://books.google.co.in/books?id=Np9SDQAAQBAJ.

T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *2010 IEEE International Conference on Software Maintenance*, 2010. doi: 10.1109/ICSM.2010.5609665.

M. Kaptein and E. R. van den Heuvel. *Statistics for Data Scientists - An Introduction to Probability, Statistics, and Data Analysis*. Undergraduate Topics in Computer Science. Springer, 2022. ISBN 978-3-030-10530-3. doi: 10.1007/978-3-030-10531-0. URL https://doi.org/10.1007/978-3-030-10531-0.

W. E. Lewis and W. H. C. Bassetti. *Software Testing and Continuous Quality Improvement, First Edition*. Auerbach Publications, USA, 2000. ISBN 0849398339.

M. L. R. Marinescu. *Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

S. Marsland. *Machine Learning - An Algorithmic Perspective*. Chapman and Hall / CRC machine learning and pattern recognition series. CRC Press, 2009. ISBN 978-1-4200-6718-7.

P. Muhammad Ali and R. Faraj. Data normalization and standardization: A technical report. 01 2014. doi: 10.13140/RG.2.2.28948.04489.

K. P. Murphy. *Machine learning : a probabilistic perspective*. MIT Press, Cambridge, Mass. [u.a.], 2012. ISBN 9780262018029 0262018020. URL https://www.amazon.com/Machine-Learning-Probabilistic-Perspective-Computation/dp/0262018020/ref=sr_1_2?ie=UTF8&qid=1336857747&sr=8-2.

K. P. Murphy. *Machine learning : a probabilistic perspective*. MIT Press, Cambridge, Mass. [u.a.], 2013. ISBN 9780262018029 0262018020. URL https://www.amazon.com/Machine-Learning-Probabilistic-Perspective-Computation/dp/0262018020/ref=sr_1_2?ie=UTF8&qid=1336857747&sr=8-2.

S. Olbrich, D. Cruzes, and D. Sjøberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. pages 1–10, 09 2010. doi: 10.1109/ICSM.2010.5609564.

T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant'Anna. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development*, 5, 12 2017. doi: 10.1186/s40411-017-0041-1.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

N. E. F. . S. L. Pfleeger. *Software metrics: A rigorous and practical approach. Third edition.* Taylor & Francis group, 2015.

C. Sammut and G. I. Webb, editors. *Mean Squared Error*, pages 653–653. Springer US, Boston, MA, 2010. ISBN 978-0-387-30164-8. doi: 10.1007/978-0-387-30164-8_528. URL https://doi.org/10.1007/978-0-387-30164-8_528.

T. Sharma and M. Kessentini. Qscored: A large dataset of code smells and quality metrics. *MSR2021*, 01 2021. doi: 10.1109/MSR52588.2021.00080. URL https://par.nsf.gov/biblio/10300802.

S. Vidal, I. n. berra, S. Zulliani, C. Marcos, and J. A. D. Pace. Assessing the refactoring of brain methods. *ACM Trans. Softw. Eng. Methodol.*, 27(1), apr 2018. ISSN 1049-331X. doi: 10.1145/3191314. URL https://doi.org/10.1145/3191314.

H. P. Vinutha, B. Poornima, and B. M. Sagar. Detection of outliers using interquartile range technique from intrusion dataset. In S. C. Satapathy, J. M. R. Tavares, V. Bhateja, and J. R. Mohanty, editors, *Information and Decision Sciences*, pages 511–518, Singapore, 2018. Springer Singapore. ISBN 978-981-10-7563-6.

J. Visser, S. Rigal, R. van der Leek, P. van Eck, and G. Wijnholds. *Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code*. O'Reilly Media, Inc., 1st edition, 2016. ISBN 1491953527.

Appendices

A The ER-digram of Qscored

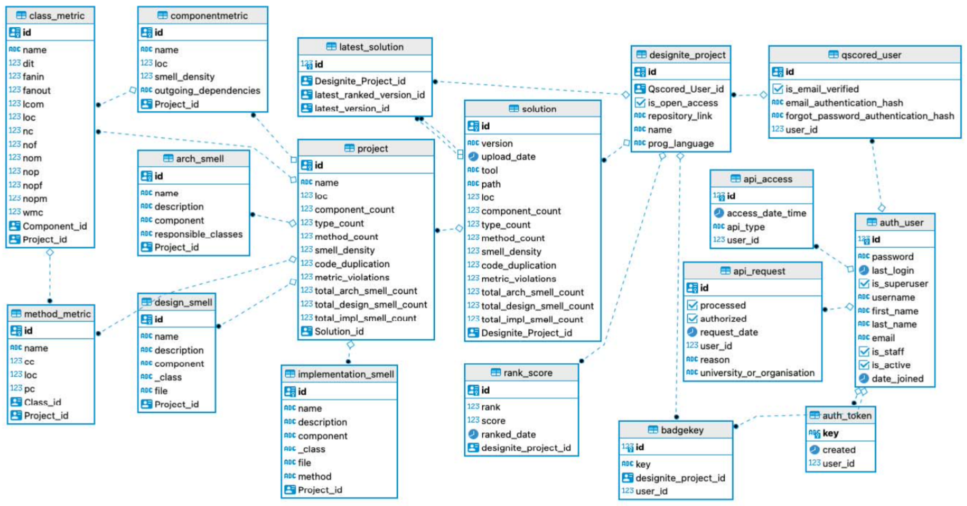


Figure 23: Caption

B Open sources files which were compared

The links to the open source projects in section 4.3.

Project number	File name	Hyperlink
1	ecomms	https://github.com/objectthink/ecomms
2	glazedlists-tutorial	https://github.com/glazedlists/glazedlists-
3	StjMbs	https://github.com/cemkuru/StjMbs
4	WeatherService	https://github.com/NickTaSpy/WeatherService

C Code to get cyclomatic complexity

Create table temp:

```
CREATE TABLE temp AS
    (SELECT project.id AS
   project_id,latest_solution."latest_ranked_version_id" AS
solution_id,latest_solution."Designite_Project_id"
FROM latest_solution
INNER JOIN project ON project."Solution_id" =
latest_solution.latest_ranked_version_id);
```

Get cyclomatic complexity.

```
SELECT SUM(method_metric.cc) AS cc, temp."solution_id"
FROM method_metric
INNER JOIN temp ON method_metric."Project_id" =
temp."project_id" GROUP BY temp."solution_id";
```

D The form for developers to evaluate and rank real world files.

Hi!

First of all, thank you for taking the time to help us with our thesis! We are two students at LTH who are trying to evaluate code quality using machine learning algorithms. The thesis goes by the working title *Measuring code quality using machine learning algorithms* and will be published in early summer 2022.

To properly see the performance of the model, we need to compare it with real life appliances. Therefore, it will be very valuable to get the opinions of experienced developers and to compare them to our model to see if they agree in terms of the quality of the code.

Instructions: We have attached four front-end and four back-end files that we would like you to review separately. The goal is to assess these files in terms of code quality.

You **should** consider aspects like the complexity of the code, if everything is robustly implemented, if it would take a lot of time to maintain the code, if the code might be error prone etc.

You **should not** consider things like, how many people have been working on the file, if the person who worked on this file does not work here anymore. You should strictly focus on the logic and how well the code is written.

---------------- **Front-end** --------------------

File 1:

Q: What did you think about the code quality in this file?
A:

File 2:

Q: What did you think about the code quality in this file?
A:

File 3:

Q: What did you think about the code quality in this file?
A:

File 4:

Q: What did you think about the code quality in this file?
A:

Q. Finally, rank each of these front-end files individually from 'best' to 'worst' with respect to code quality.

Best

File no: ______________
File no: ______________
File no: ______________
File no: ______________

Worst

Q: Give an insight to why the files were ranked liked this? What was your reasoning?
A:

----------------- **Back end** --------------------

File 1:

Q: What did you think about the code quality in this file?
A:

File 2:

Q: What did you think about the code quality in this file?
A:

File 3:

Q: What did you think about the code quality in this file?
A:

File 4:

Q: What did you think about the code quality in this file?
A:

Q. Finally, rank each of these back-end files individually from ‘best’ to ‘worst’ with respect to code quality.

Best

File no: _______________
File no: _______________
File no: _______________
File no: _______________

Worst

Q: Give an insight to why the files were ranked liked this? What was your reasoning?
A:

----------- The end ------

Once again, we are grateful for the inputs as it will massively help with writing our thesis and validating the model.

If you have any questions, please contact us at

erik.kindt@ingka.com
arvid.malmstrom@ingka.com

Best regards,

Erik & Arvid

E Comments on files from developers.

E.1 Front end

Table 20: File 1: comments

Developer	Comment about code quality
Developer 1	• Complexity: High. High nesting. Nesting of JSX and functions. • Robustness: Low. Hard to see. • Maintainability: Low. Commented out old code. Comments which do not make sense. Could use more types. Disabled of Eslint rules. Use of exclamation mark. • Readability: Low. Hugh file with mixed JSX and functions. Not using async/await. Unused parameters to functions. Unnecessary checks e.g. of profileData.profile • Error prone: Possibly. Too hard to see.
Developer 2	Bad code quality. The logic and internal states are not clear what they do. At least not on a first glance. It also have too many responsibilities. Style inline as well as from separate file.
Developer 3	Code is complex and difficult to maintain.

Table 21: File 2: comments

Developer	Comment about code quality
Developer 1	• Medium. Nesting of JSX and functions. Many useEffects. • Robustness: Medium. Seems more robust than above file. • Maintainability: Medium. Large file. Disabled of Eslint rules. Duplication of code. • Readability: Medium. Hugh file with mixed JSX and functions. Not using async/await. Some effects could be refactored out. Util function in file. • Error prone: Possibly. Easier than above to see.
Developer 2	Same as file 1 too many states. Hard to follow what happens in states. Es-lint disabled for unknown reasons. Hard to extend with new functionality.
Developer 3	Code is complex and difficult to maintain.

Table 22: File 3: comments

Developer	Comment about code quality
Developer 1	• Complexity: Low. Large switch statement but easy to follow. Nested switch statements, could be broken out to functions to reduce complexity. • Robustness: High. Seems to handle the unexpected. • Maintainability: High. Has quite some duplications which might be able to remove. • Readability: High. Nice with render/functions for each size. • Error prone: Does not look to be.
Developer 2	Cleaner, although a very long file it is at least quite clear what the file does. Some weird hook that handles form triggers is disturbing the peace.
Developer 3	This file has a good structure and it is easier to maintain.

Table 23: File 4: comments

Developer	Comment about code quality
Developer 1	• Complexity: Low. Nice with own functions for rendering different, but quite some duplicated code. • Robustness: High. Seems to handle the unexpected. • Maintainability: High. Has quite some duplications which might be able to remove. • Readability: High. Nice with render/functions for each size. • Error prone: Does not look to be.
Developer 2	Also reasonably good. Although could be separated into more files. Since too many components are added in the file entry.
Developer 3	This file has a quite good structure and it is easier to maintain.

E.2 Back end

Table 24: File 1: comments

Developer	Comment about code quality
Developer 1	• Complexity: Low. • Robustness: High. Seems to handle the unexpected. • Maintainability: Medium. A bit harder to maintain. • Readability: High • Error prone: Does not look to be.
Developer 2	Although clear what the responsibilities of the methods are, the logic and methods behind it are hard to follow and understand. Some logic could be broken out into methods but since it is a controller with methods and the methods have no common functionality maybe that would make it even harder to follow. Also too many if cases makes it hard to see what the ending result of each method is.
Developer 3	Good error handling, good documentation, good structure

Table 25: File 2: comments

Developer	Comment about code quality
Developer 1	• Complexity: Low. One nested switch statement. • Robustness: High. Seems to handle the unexpected. • Maintainability: High. Small contained functions. • Readability: High. Small contained functions. • Error prone: Does not look to be.
Developer 2	Quite straight forward. Easy simple class with methods.
Developer 3	Very good structure, missing error handling,good structure

Table 26: File 3: comments

Developer	Comment about code quality
Developer 1	• Complexity: Medium. Long file with nested if and switch statements. • Robustness: High. Seems to handle the unexpected. • Maintainability: Low. • Readability: Medium. The readability would be increased if the parameter isn't deconstructed directly. • Error prone: Does not look to be.
Developer 2	Badly written file. Although naming is good it creates a user. It has some weird exceptions depending on country and error handling is hard to understand.
Developer 3	Code is complex and difficult to maintain.

Table 27: File 4: comments

Developer	Comment about code quality
Developer 1	• Complexity: Low. One nested switch statement. • Robustness: High. Seems to handle the unexpected. • Maintainability: High. • Readability: High. Nice with render/functions for each size. • Error prone: Does not look to be.
Developer 2	Given the complexity of the of the file I think the purpose of the class is quite clear. Logic is separated into methods with good naming conventions. The file is also FW based file it has a base class that you need to understand. If you do not know how the base classes works then it is hard to understand how lifecycle methods are called.
Developer 3	Good error handling, easy to maintain, good structure