Prediction of code lifetime

Per Nordfors

Supervisor: Rita Kovordanyi
Examiner: Jose M. Peña
Upphovsrätt


Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/her own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page [http://www.ep.liu.se/](http://www.ep.liu.se/)

© Per Nordfors
Abstract

There are several previous studies in which machine learning algorithms are used to predict how fault-prone a piece of code is. This thesis takes on a slightly different approach by attempting to predict how long a piece of code will remain unmodified after being written (its “lifetime”). This is based on the hypothesis that frequently modified code is more likely to contain weaknesses, which may make lifetime predictions useful for code evaluation purposes. In this thesis, the predictions are made with machine learning algorithms which are trained on open source code examples from GitHub. Two different machine learning algorithms are used: the multilayer perceptron and the support vector machine. A piece of code is described by three groups of features: code contents, code properties obtained from static code analysis, and metadata from the version control system Git. In a series of experiments it is shown that the support vector machine is the best performing algorithm and that all three feature groups are useful for predicting lifetime. Both the multilayer perceptron and the support vector machine outperform a baseline prediction which always outputs the mean lifetime of the training set. This indicates that lifetime to some extent can be predicted based on information extracted from the code. However, lifetime prediction performance is shown to be highly dataset dependent with large error magnitudes.
Acknowledgments

I would like to thank Combitech AB for offering this thesis opportunity and the helpful team at Reality Labs for lending their support throughout the working process. Special thanks to Pablo Karlsson for suggesting an interesting subject and for his involvement in the early stages of the project. Special thanks also to Ekhiotz Vergara for his valuable advice on report writing.

In addition, I would like to thank my university supervisor Rita Kovordanyi and examiner Jose M. Peña for their feedback on the work and for useful discussions of the task.

Finally, I wish to express my gratitude to my friends and family for their encouragement.
Contents

Abstract iii
Acknowledgments iv
Contents v
List of Figures vii
List of Tables viii

1 Introduction 2
  1.1 Aim ................................................................. 2
  1.2 Methodology .................................................... 3
  1.3 Delimitations ................................................... 3
  1.4 Related work .................................................... 4
  1.5 Thesis outline .................................................. 4

2 Theory 5
  2.1 Git ................................................................. 5
    2.1.1 Commits ..................................................... 5
    2.1.2 Branches .................................................... 6
    2.1.3 Commands ................................................... 7
  2.2 Source code features .......................................... 7
    2.2.1 Code contents .............................................. 7
    2.2.2 Static code analysis ..................................... 8
    2.2.3 Change metadata .......................................... 8
  2.3 Preprocessing of features .................................... 9
  2.4 The multilayer perceptron ................................... 9
    2.4.1 Basic concept ............................................. 9
    2.4.2 Backpropagation ......................................... 11
    2.4.3 Early stopping ........................................... 12
    2.4.4 Network design ........................................... 13
  2.5 Support vector machines .................................... 14
    2.5.1 Basic concept ............................................. 14
  2.6 Evaluation of machine learning algorithms .................. 17
    2.6.1 Root mean squared error ................................. 17
    2.6.2 Data partitioning ........................................ 17
    2.6.3 k-fold cross-validation .................................. 17

3 Method 19
  3.1 Defining lifetime prediction ................................ 19
    3.1.1 Pieces of code ........................................... 19
    3.1.2 Lifetime .................................................. 20

v
# List of Figures

2.1 A Git commit ............................................................. 6  
2.2 Git branches .......................................................... 7  
2.3 The bag-of-words model ............................................. 8  
2.4 The multilayer perceptron ......................................... 9  
2.5 A single neuron ...................................................... 10  
2.6 Activation functions ............................................... 11  
2.7 Overfitting ............................................................ 12  
2.8 Early stopping ....................................................... 13  
2.9 $\epsilon$ and slack variables ....................................... 15  
2.10 4-fold cross-validation ........................................... 18  

3.1 Pieces of code ......................................................... 20  
3.2 Lifetime examples .................................................. 21  
3.3 Git branches ........................................................ 21  
3.4 Mapping PMD output to code ..................................... 26  
3.5 Experimental overview ............................................. 28  

4.1 Empirical CDF of lifetime ......................................... 31  
4.2 Performance on data from all repositories ..................... 37  
4.3 Performance on Retrofit ........................................... 38  
4.4 Performance on Hystrix ............................................ 39  
4.5 Performance on Gitblit ............................................. 39  
4.6 Performance on Dropwizard ...................................... 40  
4.7 Performance on Okhttp ............................................. 40  
4.8 Best performances .................................................. 41
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>GitHub repositories</td>
<td>22</td>
</tr>
<tr>
<td>4.1</td>
<td>Extracted pieces of code</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>Mean lifetime and standard deviation</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>Extracted features</td>
<td>32</td>
</tr>
<tr>
<td>4.4</td>
<td>Results from different hidden layer sizes, single feature groups</td>
<td>32</td>
</tr>
<tr>
<td>4.5</td>
<td>Results from different hidden layer sizes, combinations of feature groups</td>
<td>32</td>
</tr>
<tr>
<td>4.6</td>
<td>Grid search for $C_1$</td>
<td>33</td>
</tr>
<tr>
<td>4.7</td>
<td>Grid search for $C_{10}$</td>
<td>33</td>
</tr>
<tr>
<td>4.8</td>
<td>Grid search for $C_r$</td>
<td>33</td>
</tr>
<tr>
<td>4.9</td>
<td>Grid search for $S$</td>
<td>34</td>
</tr>
<tr>
<td>4.10</td>
<td>Grid search for $M_0$</td>
<td>34</td>
</tr>
<tr>
<td>4.11</td>
<td>Grid search for $M_1$</td>
<td>34</td>
</tr>
<tr>
<td>4.12</td>
<td>Grid search for $M_{10}$</td>
<td>34</td>
</tr>
<tr>
<td>4.13</td>
<td>Grid search for $C_{10}S$</td>
<td>35</td>
</tr>
<tr>
<td>4.14</td>
<td>Grid search for $C_rS$</td>
<td>35</td>
</tr>
<tr>
<td>4.15</td>
<td>Grid search for $C_{10}M_{10}$</td>
<td>35</td>
</tr>
<tr>
<td>4.16</td>
<td>Grid search for $C_rM_{10}$</td>
<td>35</td>
</tr>
<tr>
<td>4.17</td>
<td>Grid search for $SM_{10}$</td>
<td>36</td>
</tr>
<tr>
<td>4.18</td>
<td>Grid search for $C_{10}SM_{10}$</td>
<td>36</td>
</tr>
<tr>
<td>4.19</td>
<td>Grid search for $C_rSM_{10}$</td>
<td>36</td>
</tr>
<tr>
<td>A.1</td>
<td>Frequent code terms</td>
<td>51</td>
</tr>
<tr>
<td>A.2</td>
<td>Frequent PMD warnings</td>
<td>52</td>
</tr>
<tr>
<td>A.3</td>
<td>Frequent log message terms</td>
<td>53</td>
</tr>
</tbody>
</table>
Introduction

As a lot of today's society relies on software, the role of software development is of great importance. Considering the effort and time spent on developing large amounts of source code every day, optimizations of the development process could lead to large time savings. Among such optimizations, analyzing newly written code in order to discover and avoid potential bugs or unoptimized expressions is interesting. This can be used for suggesting code to be rewritten at an early stage, thus avoiding infeasible or difficult code revisions later on in the process. For this purpose, there are several useful tools for static code analysis, which automatically scan the code for weaknesses and report their findings to the developer.

By using version control systems to access the revision history of source code projects, properties of code that is known to have induced faults can be extracted. Such information can be used for training machine learning algorithms to recognize potentially problematic code, as suggested by Mockus and Weiss [1], Kim, Whitehead Jr., and Zhang [2] and Snipes, Robinson, and Murphy-Hill [3]. Today, numerous large open-source projects along with their revision history can be accessed publicly via the web, which makes data for such machine learning tasks easy to obtain.

While several previous studies [1, 2, 3] attempt to use machine learning to predict how fault-prone a piece of code is, a slightly different approach is adopted in this study. Considering code lifetime as the time for which code remains unmodified, this work explores if it is possible to predict the lifetime for a piece of code. It can be argued that source code with short lifetime is likely to contain weaknesses and predictions of lifetime on newly written code may therefore have an effect on the behavior and decisions made when writing code, similarly to predictions of potential faults. In a real-life setting, a developer would likely be interested in evaluating the latest contributions of code. Therefore, the lifetime predictions will be designed to be applied on newly written code, based on information that is available at the time the code is written.

1.1 Aim

The aim of this study can be concluded in the following research questions:

1. How precisely can the lifetime of a piece of code be predicted?
1.2 Methodology

Using machine learning models, predictions of lifetime will be made for pieces of code. Lifetime is represented as a number of discrete timesteps. Formal definitions of lifetime and pieces of code are introduced later on in section 3.1. The evaluation of the prediction performance is described in section 2.6. Based on the accuracy of the predictions, it will be discussed how useful these predictions could be in practice. In this study, two different machine learning algorithms will be used, which gives rise to the following sub-question:

1a. How do the multilayer perceptron (MLP) and support vector machine (SVM) compare to each other on the given task?

MLP and SVM are commonly used machine learning algorithms in regression problems. In such problems, a trained model acts as a function that is designed to fit the data in a given dataset as closely as possible. The model can then be used to produce a numerical output for new input data. In the context of lifetime prediction, which is here regarded as a regression problem, the desired function maps a piece of code (described by a set of features) to a lifetime value. In this study, the performance of the MLP and SVM algorithms will be compared.

2. What different features can be useful for such predictions?

The features used for predicting the lifetime will be extracted both from the code itself and its revision history. The influence of different feature groups on the results will be investigated as it is not certain that all information that can be extracted contributes to improving the prediction performance.

1.2 Methodology

The overall solution will consist of constructing a system that uses data from the version control system Git to predict lifetime for pieces of code, based on information that is available at the time the code is written. The data will be collected from five different open-source projects available at GitHub [4] (an online Git host). The solution can be divided into the following steps:

1. Define lifetime and pieces of code. The definitions will be based on what would be useful if lifetime prediction was used in practice and what is convenient for this study.
2. Extract pieces of code and calculate their lifetime from Git. This will done by traversing the revision history for every file in a Git repository. The set of code pieces with known lifetime values will be used for training and evaluating the machine learning models.
3. Extract features for the pieces of code. These will be extracted either from Git directly or by analyzing the file that contains the piece. The choice of features will be based on other works concerning prediction on source code.
4. Conduct experiments for selecting design parameters for the machine learning algorithms. The parameters which yield the best results are used to produce a set of final models.
5. Measure the prediction performance of the final models on unseen data. The experiments will be conducted using different feature groups and datasets.

1.3 Delimitations

In this study, only source code written in the Java language is considered. The study is also delimited to using Git as the only version control system.
The code used as data only includes code that has actually been modified after being written (i.e. the time it remains unmodified is known). In the 5 repositories used in this study, this comprised between 48% and 65% of the code pieces.

No other machine learning algorithms than MLP and SVM are used. The feasibility of solving this prediction task is therefore based on the performance of these models only.

1.4 Related work

There are several published examples in which some code quality aspect is predicted in order to aid the software development process. In many cases, the presence of potential faults is the quality of interest. This was deemed related to the subject of this study (prediction of code lifetime) based on a hypothesis that code that contains faults or weaknesses is related to code that is modified. Consequently, lifetime prediction was approached similarly to such tasks. In previous work related to fault prediction, three interesting factors can be identified: what is predicted (i.e. what the output value represents), for which code pieces predictions are made, and what features are used to describe the code.

Regarding what is predicted, Kim, Whitehead Jr., and Zhang [2] as well as Snipes, Robinson, and Murphy-Hill [3] output a class label depending on whether the code contains potential faults. Mockus and Weiss [1] predict the risk (a probability value) of code changes causing future faults. Another form of prediction is shown by Gyimóthy, Ferenc, and Siket [5], who predict the number of bugs in the code. As this study focuses on predicting the lifetime for a piece of code, the output value is the respect in which it differs largely from previous work.

As for which code pieces to make predictions for, several different approaches have been taken in previous work. One variant consists of considering all the code that has been changed in a submission to the version control system [1, 2]. Another variant is to make prediction for individual source code files [3]. Predictions can also be made for certain structures in the code, such as classes [5] or functions, as done by Menzies, Greenwald, and Frank [6].

In the above examples, different kinds of features are used to describe the code for which to make a prediction. In several cases, metadata extracted from the version control system have been used [1, 2, 3]. Also code complexity metrics (e.g. lines of code, cyclomatic complexity) have been proven to be useful features for fault prediction [6]. For predicting the number of bugs in a class, object-oriented metrics can be used as features [5]. Even though textual code contents may intuitively seem useful, this information is not widely used for fault prediction tasks. It has, on the other hand, been shown to be useful for classifying code changes with respect to potential faults [2].

1.5 Thesis outline

Chapter 2 provides an overview of related work and theoretical concepts that were adopted in the study. In chapter 3, the different parts of the solution are explained, i.e. the definitions of lifetime and code pieces, extraction of data, selection of MLP and SVM design parameters and measuring the lifetime prediction performance of the final models. The results from the data extraction and the experiments are presented and analyzed in chapter 4. Further discussions of the method and results are provided in chapter 5. Finally, chapter 6 presents the conclusions of the study and discusses the value of predicting code lifetime.
This chapter provides a theoretical background for some of the concepts adopted in the study. Section 2.1 explains the basic concepts of the version control system Git, which was used for extracting data for the study. Section 2.2 gives an overview of different feature groups that can be used for describing source code. Section 2.3 explains how the features were preprocessed. The two machine learning algorithms used in the study – MLP and SVM – are presented in sections 2.4 and 2.5 respectively. Section 2.6 describes a few concepts related to evaluation of machine learning algorithms.

2.1 Git

Git [7] is a distributed version control system that allows multiple developers to share and contribute code in software development projects. Since the data used in this project was extracted from Git repositories, a basic explanation of a couple of important Git concepts is given in this section.

2.1.1 Commits

Typically, a developer who works on a project keeps all the code related to the project in a Git repository. When working, changes are made to the repository and a check-in of a set of changes is represented by a commit which contains a snapshot of the repository contents at the time of the check-in. It also contains metadata, such as the name of the committing developer, commit time and date, and a log message written by the developer to describe the set of changes [8]. For each commit, Git also keeps track of all lines of code that have been added or deleted.

Figure 2.1 shows some of the information that can be displayed about a commit by a Git user. This includes a commit id, the name of the committing developer, time and date of the commit, a log message. Below, the code sections that have been modified since the previous commit are grouped together in so called hunks. In this example, the hunks contain only consecutive lines of modified code, but the amount of surrounding code can be specified with certain options (see section 2.1.3). Each hunk starts with a line on the format

```bash
@@ -a,b +c,d @@
```
2.1. Git

where \( a \) indicates the start of a hunk (line number) according to what the file looked like in the previous commit, and \( b \) indicates the number of lines affected by the change (this number is omitted if one single line is affected). \( c \) indicates where this hunk starts in the current version of the file and \( d \) is the number of lines added in the current version, which is also omitted in the case of a single line. Added and deleted lines of code in each hunk are marked with “+” and “-“, respectively. Git makes no distinction between modified and deleted lines – they are both marked with “-”.

Simplified, the revision history of a repository can be viewed as a series of commits. However, this is usually not the case, which is explained in the following section.

commit f4c0757a9c1fd14b570c9bf957f15de271c4bcl
Author: Per Nordfors <pelle_nordfors@mail.com>
Date: Tue Aug 2 14:25:24 2016 +0200

Rewrote some stuff and added a new function.

diff --git a/subfolder/hello.java b/subfolder/hello.java
index 4d42f73..70ae8ae 100644
--- a/subfolder/hello.java
+++ b/subfolder/hello.java
@@ -4,2 +4,2 @@ public class HelloWorld {
-    // Prints "Hello, Universe" to the terminal window.
-    System.out.println("Hello, Universe");
+    // Prints "Hello, World" to the terminal window.
+    System.out.println("Hello, World");
@@ -8,2 +8,2 @@ public class HelloWorld {
-    public int two() {
-        return 2;
+    public int one() {
+        return 1;
@@ -12 +12,3 @@ public class HelloWorld {
-        // A function should be here:
+        public void doNothing() {
+            // Does nothing...
+    }

Figure 2.1: Information describing a commit. Code changes are displayed in several hunks.

2.1.2 Branches

Git allows multiple developers to work on code in the same repository. Since the developers may be working on different parts of the same system, it is customary to use different so-called branches, which produce parallel lines of commits which can be merged when desired. Figure 2.2 shows an example of two branches. The master branch represents the main line of commits while another branch for testing out a new feature creates a parallel line of commits. When the new feature is ready to be added to the main line of the project, this branch merges with the master branch. A consequence of this is that commits made after the merge operation will have multiple “paths” of ancestors. [8]
2.2 Source code features

Figure 2.2: Development work progresses in two parallel branches, where each circle represents a commit. The new feature branch initially branches out from the master branch and the two are merged at a later stage. The last (rightmost) commit thus has two paths of ancestor commits.

2.1.3 Commands

Two Git commands that can be used for displaying information about the revision history of a project are `log` and `diff`. Combined with different options, the commands can display more specific information.

The `log` command prints a list, containing short summaries of all changes made to a repository. A couple of options are:

- `--follow` – Prints only commits related to a specific file.
- `--pretty` – Prints only information that is specified by an argument, such as the committing developer, commit id or log message.

The `diff` command can be used to compare two commits, which results in a list of hunks, similar to the one seen in figure 2.1. The following options can be used:

- `--follow` – Compares two versions of a specific file.
- `--unified` – Combined with a numerical argument, this specifies the amount of code that is included in a hunk in addition to the consecutive lines of modified code.

2.2 Source code features

Many of the features used in the works mentioned in 1.4 can be considered relevant for life-time prediction, as they describe different code characteristics. Three feature groups can be formed based on the source of information and what the values represent: code contents, static code analysis, and change metadata.

2.2.1 Code contents

Kim, Whitehead Jr., and Zhang [2] extract features describing code contents with a bag-of-words model, which is used to create a vector representation of textual information. In one such vector, each index corresponds to a term in a vocabulary that consists of all terms encountered in texts in the dataset [9]. The value at each index, which represent one feature, can be set in multiple ways in order to reflect the contents of the text. A simple solution is to either use binary values (0 or 1) indicating whether or not the term is present in the text, or the term frequency, i.e. the number of occurrences of the term [10]. Figure 2.3 shows an example of a line of code and its corresponding vector based on term frequency.

Note that using this model for source code disregards the code structure. On the other hand, it is easy to implement and still provides information about the terms present in the code.
2.2. Source code features

One consequence of the bag-of-words model is that the feature vector grows large with a high number of unique terms in the vocabulary, which may significantly increase the time for training machine-learning models such as MLP or SVM. This issue can be remedied by excluding terms that are likely to provide little useful information for the prediction task.

In text-categorization tasks, where text is divided into categories based on content, a subset of the vocabulary containing only the most frequent terms can be used to characterize text. A subset containing only the 10% most frequent terms can suffice for text categorization tasks without any notable loss in performance. Even 1% subsets may cause only small losses. This is possible due to the fact that textual information that is most essential for characterizing a text is usually described in relatively frequent terms. The terms with very low term frequency (which is generally the case for most of the terms) are usually not important for this purpose [11].

2.2.2 Static code analysis

By using static code analysis tools, complexity metrics for a code entity (typically a file) can be obtained. Examples of such metrics are the number of decision points in the code (also known as cyclomatic complexity), the number of statements in methods, and the number of method parameters. Complexity metrics can be used as code features in fault-prediction tasks, as shown by Menzies, Greenwald, and Frank [6] and Kim, Whitehead Jr., and Zhang [2]. This is motivated by the fact that code with high complexity is generally more likely to contain faults [6].

2.2.3 Change metadata

Version control systems like Git provide a database of changes made to the source code in a project and multiple different properties can be extracted as features. Some features can be extracted directly from a specific commit, such as who the committing developer is, the number of added/deleted lines in the commit and the number of lines in the modified files. Log messages written by the committing developer to describe the commit can also be used as features, represented e.g. in bag-of-words form (as described in section 2.2.1) [2].

Another type of features that can be extracted from the version control system concerns the diffusion of commits. Examples of such features are the number of system parts that have been modified in a commit and the number of developers who have contributed to a commit or a file. The use of this type of features is motivated by the fact that changes are more likely to be faulty as the diffusion increases [1][3].

Other features include the number of times a file has been modified at the time of a commit and developer experience. Experience is measured by counting the number of previous commits by the committing developer. As shown by Mockus and Weiss [1], more experienced developers are less likely to induce faults.

Figure 2.3: A line of code (left) and its corresponding vector (right) when using a bag-of-words model.
2.3 Preprocessing of features

As explained in section 2.2, features can be extracted from multiple sources and represent different kinds of values, such as code complexity metrics or term frequencies in a bag-of-words model. The diversity of the variables is not necessarily a problem, as there are published examples using mixed-type variables for both artificial neural networks (ANN) (which MLP is a form of) [12, 13, 14] and support vector machines (SVM) [15]. It may, however, put demands on data preprocessing.

ANN benefit from having the values of each feature centered around zero, as it speeds up the convergence of training [16]. Additionally, both ANN and SVM benefit from having the feature values in approximately the same range, [13, 16, 15, 17]. This prevents features with large ranges of values from dominating over features with smaller values in the training process. If the features were scaled very differently, those with large ranges of values would have a bigger influence on the result, thus making them seem more important, which they may not be.

One way to accomplish scaling and centering around zero is to replace every variable \( x \) in the feature vector with its standard score \( z \), which is calculated from

\[
    z = \frac{x - \mu}{\sigma}
\]

where \( \mu \) is the variable’s mean in the dataset and \( \sigma \) is its standard deviation. The standard deviation for each variable in the feature vector will thereafter be 1 and its mean will be 0. This normalization technique can be used for both ANN [18] and SVM [17].

2.4 The multilayer perceptron

The multilayer perceptron (MLP) is a machine learning algorithm which is a form of artificial neural networks (ANN). These can be used for a wide array of tasks, spanning from image recognition to simulation of electronic components. As lifetime prediction is regarded as a regression task, this section focuses mainly on how MLP can be applied for such tasks. A property that makes MLP useful for regression is the ability to learn highly complex mappings between input and output.

2.4.1 Basic concept

MLP consist of a network of nodes called neurons. Each neuron node in the network is a processing unit whose main task is to calculate an output from its inputs and forward it to subsequent neurons (see figure 2.4).

![Figure 2.4: MLP with three layers of neurons: input layer (left), hidden layer (middle) and output layer (right).](image)

The MLP neurons are organized into layers, such that neurons in one layer receive input from the preceding layer and produce outputs which serve as inputs to the next layer. MLPs are fully connected, meaning that the output of a neuron is forwarded to every neuron in the next layer. The network has one input layer, one output layer and one or many hidden
2.4. The multilayer perceptron

layers in between. The input layer has a size (number of neurons) that equals the size of the feature vector used as input to the network. The neurons in the input layer only have one input, which corresponds to one element of the feature vector. In regression tasks, where a numerical output value is desired, the output layer consists of only one neuron.

The calculation of output values from neurons in the hidden and output layers consists of two main operations: summing their inputs and applying an activation function to the sum. As the input edges to a neuron are weighted, each input is multiplied by an edge-specific weight coefficient. For a neuron with inputs, the calculation of the output value can be written as

\[ b_j = \theta(\sum_{i=1}^{l} w_{ij}x_i) \]  \hspace{1cm} (2.2)

where \( \theta \) is the activation function for \( j \) and \( w_{ij} \) is the weight of the edge between a neuron \( i \) and \( j \). A close-up of a single neuron is depicted in figure 2.5 for clarity.

Figure 2.5: A single neuron receiving inputs and calculating an output value.

In the single neuron output layer for regression tasks, a linear activation function is used [12], meaning that the output is a linear combination of the inputs with the coefficients given by the weights of the input edges:

\[ \theta(\sum_{i=1}^{l} w_{ij}x_i) = \sum_{i=1}^{l} w_{ij}x_i \]  \hspace{1cm} (2.3)

In the hidden nodes, nonlinear activation functions can be used, which make it possible for the network to approximate nonlinear functions [18, 12]. Frequent choices of such activation functions include the logistic function (equation 2.4) and the hyperbolic tangent (tanh) (equation 2.5).

\[ f(x) = \frac{1}{1 + e^{-x}} \]  \hspace{1cm} (2.4)

\[ tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \]  \hspace{1cm} (2.5)

The hyperbolic tangent may be preferred in order to keep the neuron outputs close to and symmetric around 0 (combined with properly normalized data), which allows weights to be updated in different directions with respect to a single input during training [16]. The function curves for the above functions are displayed in figure 2.6.
2.4. The multilayer perceptron

2.4.2 Backpropagation

Backpropagation is the training algorithm for MLP. The objective of training an artificial neural network such as MLP is to set the values of the weights in the network such that the output errors (i.e., the difference between predicted values and correct values) are minimized. Usually, a subset of the total dataset is used for training (the training set) while the rest of the data (the test set) is reserved for evaluating network performance on previously unseen data. Data partitioning is described further in section 2.6.

Before the training starts, the weights are initialized with random values (positive and negative) with a zero mean [19]. During training, the network is given an input from the training set and produces an output. The network’s output $y$ is then evaluated with respect to the correct value $t$ by a cost function, which usually is based on the mean squared error [12]:

$$E = \frac{1}{2}(t - y)^2$$

(2.6)

The next step is to modify the weights in a direction that minimizes the error. This is done by first calculating the influence of each weight on the network output (or more specifically, the partial derivative of the error with respect to each weight) and then modifying the weights proportionally to their influence. This procedure is performed layer-wise, starting from the output layer of the network. After calculating the partial derivatives of the error with respect to each of the weights in the hidden layer closest to the output, these can be used (by applying the chain rule) to express the partial derivatives of the error with respect to the weights of next layer, and so on. This is repeated until the partial derivatives with respect to all weights in the network (denoted as matrix $w$) have been calculated, giving $\frac{\partial E}{\partial w}$. In order to minimize the error, the weights are updated in the opposite direction of their error contribution, resulting in new weights

$$w_{new} = w - \eta \frac{\partial E}{\partial w}$$

(2.7)

where $\eta$ is the learning rate. This procedure, known as backpropagation, is repeated until a stopping criterion is met, e.g., reaching a maximum number of epochs (iterations over the entire training set) or the average error over one epoch falling below a limit [20]. Optionally, early stopping (described in section 2.4.3) can be implemented.

The updating of weights can be seen as an optimization problem, where a minimal cost with respect to the cost function is desired. Neural networks can, however, find local minima
which are far from globally optimal. This can be helped by using a momentum when updating the weights. Equation \(2.8\) shows how \(\Delta w\) (corresponding to the difference between \(w_{\text{new}}\) and \(w\) in equation \(2.7\)) is calculated with a momentum \(m\) in the \(n\):th weight update of the training procedure.

\[
\Delta w_n = m \Delta w_{n-1} - \eta \frac{\delta E}{\partial w_{n-1}}
\]  

(2.8)

The momentum specifies how big the influence of the previous update (\(\Delta w_{n-1}\)) should be when calculating the new weights, which reduces the immediate influence of the most recent error. Using a momentum can also make the network training converge faster \([18, 16]\).

Updating the weights after each input with respect to the output error is called online learning or stochastic learning. An alternative approach is to calculate the cost after processing the entire training set and modify the weights based on the total error. This approach is called batch learning. Online learning is faster than batch learning and often results in better solutions \([16, 18]\).

When the training phase is completed, the network performance is evaluated with previously unseen data. The evaluation step is described further in section \(2.6\).

### 2.4.3 Early stopping

As described in the previous section, a maximum number of epochs or a lower limit of the error can be used as stopping criteria, which leads to a potentially large number of training epochs. However, the performance on the test set does not necessarily benefit from a high number of training epochs. Actually, the performance can get worse if the network reaches a point where it, instead of learning from the patterns in the training data, starts to learn its specific characteristics. This is called overfitting \([18]\).

Ideally, one wants to stop training at the point where the performance on the test set would be the best (see figure \(2.7\)). The test set must however not be used until the network is fully trained, for the purpose of measuring its performance on unseen data (otherwise, the data is not unseen). Therefore, there is no way of knowing when training should be stopped in order to achieve its best possible performance on the test set, but this can be approximated by implementing an early stopping technique.

![Figure 2.7: The error decreases quite steadily for the training set (black) as the number of epochs increases, while the error on the test set (green) starts increasing after reaching a minimum (dashed line). If training continues after the error on the training set starts to increase, overfitting occurs.](image)
A first step is to divide the training data into two sets: one training set and one validation set. The training set, like before, is used to train the network in each epoch. The validation set, however, acts as an “unseen” data set and is used to validate the network in regular intervals during training in order to approximate the performance on the test set. The idea is to stop training when the error on the validation set (the validation error) has reached a minimum. Since a minimum cannot be recognized as such until the validation error has started to increase again, the states of the network are recorded during training. When encountering an increase in the validation error, training is stopped and the weights of the network recorded at the point of the minimum are selected.

In practice, the error curves are usually noisy and contain numerous local minima, as shown in figure 2.7. To stop training as soon as a minimum is encountered may therefore be far from optimal, which motivates the use of more sophisticated stopping criteria.

One such stopping criterion can be formed by allowing the validation error to increase for a certain number of successive validations before stopping training. The intention is to disregard small, shallow local minima that arise from the noisy character of the curve but still capture the overall trend of the validation error. When the maximum number of successive increases is observed, training is stopped and the weights that gave the lowest recorded validation error are selected. An example is depicted in figure 2.8. Note that this is by no means guaranteed to be global minimum, but rather a local minimum that is “distinct enough”. Neither can it be guaranteed that this criterion will ever stop the training, which is why additional stopping criteria (e.g. a maximum number of training epochs) need to be used [20].

![Figure 2.8: The validation error (orange) has increased in three successive validations. If the limit is set to 3, training will terminate at this point and the weights at the epoch with the lowest validation error (the dashed line) will be selected.](image.png)

### 2.4.4 Network design

Apart from selecting activation functions and setting parameters related to training, a crucial part of a neural network solution is settling on the number of hidden layers as well as the number of hidden neurons to use.

Regarding the number of hidden layers, one is sufficient for approximating arbitrary non-linear functions [12], making the network a three-layer perceptron (input layer + hidden layer + output layer). One such solution may, on the other hand, require a large number of neurons in the hidden layer. There exist comparisons between MLPs with one and two hidden layers demonstrating no advantages of using an extra hidden layer in the general case [21]. A consequence of using three layers instead of four is a reduced number of neurons and thus a...
simpler network and reduced training time. In terms of training, a higher number of hidden
layers also makes training the network more difficult \[18, 21\].

There is no definite rule for how many hidden neurons a network should have, but prob-
lems of highly nonlinear nature generally benefit from more neurons. However, networks
with too many neurons may suffer from overfitting. If, on the other hand, too few neurons
are used, the network may be unable to learn from patterns in the data \[12\]. A few sug-
gested heuristics appearing in the ANN literature propose that the number of hidden neurons
should be

- \(2/3\) the total size of the input and output layers \[22\]
- between the input layer size and the output layer size \[23\]
- less than twice the size of the input layer \[24\]

Note that the above rules do not fully agree. In practice, trial and error can be employed
by adding or removing hidden neurons until no further improvements in performance are
made \[25, 26\].

2.5 Support vector machines

Support vector machines (SVMs) are machine learning algorithms which are commonly used
for similar tasks as MLP. SVM training consists of solving an optimization problem as op-
posed to the repeated backpropagation algorithm used for MLP. The training results in a
model that is globally optimal with respect to the data in the training set (similarly to MLP,
data is usually partitioned into separate sets for training and testing). This section focuses
mainly on how SVM can be applied for regression tasks, as lifetime prediction is regarded as
one such task.

2.5.1 Basic concept

As explained by Smola and Schölkopf \[27\] as well as Fletcher \[28\], the objective of SVM re-
gression is to find a function \(f(x)\) that optimal in the sense that it fits the training data as
closely as possible. The function can then be used for producing output values for previ-
sously unseen input. In the basic linear regression case, the function will be expressed on the
following form:

\[
f(x) = w \cdot x + b
\]

SVM regression can be expanded for nonlinear problems by applying a mapping \(x \rightarrow \phi(x)\), which maps the data to a high dimensional space that may be more well-suited for
fitting a function \(f(x)\) for the particular problem. Consequently, the desired function has the
following form in the nonlinear case:

\[
f(x) = w \cdot \phi(x) + b
\]

One objective in finding a function \(f(x)\) that is optimal with respect to the training data
is to maximize the margin, i.e. the perpendicular distance between \(f(x)\) and the closest data
point in the training set. This can be expressed as minimizing \(\frac{1}{2} ||w||^2\).

Furthermore, in order to minimize the deviations between \(f(x)\) and the actual target val-
ues \(y\) for the training data points, deviations larger than a certain limit \(\epsilon\) will be penalized.
Deviations which are smaller than \(\epsilon\) are deemed tolerable and will not be penalized. This can
be referred to as using an \(\epsilon\)-insensitive tube. Penalties are given by slack variables \(\xi^+\) or \(\xi^-\),
depending on which side of the tube the data points are located. Figure 2.9 shows an example
with data points inside as well as outside the tube. It also displays where the slack variables
\(\xi^+\) and \(\xi^-\) are nonzero with respect to the tube.
2.5. Support vector machines

Figure 2.9: As long as the target values $y$ are located in the tube around $f(x)$, $\xi^+$ and $\xi^-$ will be 0 and the deviations will not be penalized. Outside the tube, $\xi^+$ and $\xi^-$ are larger than 0 and the deviations will be penalized.

With these variables defined, the optimization problem of both minimizing $\frac{1}{2}||w||^2$ and the penalties for data points outside the tube can be formulated as follows:

$$
\begin{align*}
\text{minimize} & \quad \frac{1}{2}||w||^2 + C \sum_{i=1}^{L} (\xi_i^+ + \xi_i^-) \\
\text{subject to} & \quad y_i - w \cdot \phi(x_i) - b \leq \epsilon + \xi_i^+ \\
& \quad w \cdot \phi(x_i) + b - y_i \leq \epsilon + \xi_i^- \\
& \quad \xi_i^+ \geq 0 \\
& \quad \xi_i^- \geq 0 
\end{align*}
$$

(2.11)

where constant $C$ is a weight that balances the trade-off between maximizing the margin and minimizing deviations outside the tube. $i = 1 \ldots L$ where $L$ is the number of data points in the training set. Based on this, a Lagrange function of the primal problem can be formulated as shown in equation (2.12) $\alpha_i^+, \alpha_i^-, \mu_i^+$, and $\mu_i^-$ represent Lagrange multipliers which are $\geq 0$ for all $i$.

$$
L_p = \frac{1}{2}||w||^2 + C \sum_{i=1}^{L} (\xi_i^+ + \xi_i^-) - \sum_{i=1}^{L} \alpha_i^+ (\epsilon + \xi_i^+ - y_i + w \cdot \phi(x_i) + b) - \\
\sum_{i=1}^{L} \alpha_i^- (\epsilon + \xi_i^- + y_i - w \cdot \phi(x_i) - b) - \sum_{i=1}^{L} (\mu_i^+ \xi_i^+ + \mu_i^- \xi_i^-)
$$

(2.12)

In order to formulate the dual problem, the partial derivatives of $L_p$ with respect to $w$, $b$, $\xi_i^+$ and $\xi_i^-$ need to be calculated:

$$
\frac{\partial L_p}{\partial w} = 0 \Rightarrow w = \sum_{i=1}^{L} (\alpha_i^+ - \alpha_i^-) \phi(x_i)
$$

(2.13)

$$
\frac{\partial L_p}{\partial b} = 0 \Rightarrow \sum_{i=1}^{L} (\alpha_i^+ - \alpha_i^-) = 0
$$

(2.14)
2.5. Support vector machines

\[
\frac{\partial L_p}{\partial \xi_i^+} = 0 \Rightarrow C = a_i^+ + \mu_i^+ \\
\frac{\partial L_p}{\partial \xi_i^-} = 0 \Rightarrow C = a_i^- + \mu_i^-
\] (2.15)

The dual problem can then be formulated by substituting the partial derivatives into equation (2.12):

\[
\text{maximize } \sum_{i=1}^L \left( \frac{1}{2} (a_i^+ - a_i^-)(a_i^+ - a_i^-) \phi(x_i) \cdot \phi(x_i) - e \sum_{i=1}^L (a_i^+ + a_i^-) y_i(a_i^+ - a_i^-) \right)
\]

subject to

\[
\begin{align*}
0 & \leq a_i^+ \leq C \\
0 & \leq a_i^- \leq C
\end{align*}
\] (2.17)

According to equation (2.13), \( w \) in the optimal solution can be expressed as

\[
w = \sum_{i=1}^L (a_i^+ - a_i^-) \phi(x_i)
\] (2.18)

The function used for prediction on a previously unseen data points \( x' \) can then be expressed as

\[
f(x') = \sum_{i=1}^L (a_i^+ - a_i^-) \phi(x_i) \cdot \phi(x') + b
\] (2.19)

For some \( i, a_i \) will be 0, which means that the data point makes no contribution in predictions with \( f(x) \). The remaining data points, for which \( a_i > 0 \) (i.e. outside the tube), are called support vectors. By identifying the support vectors, \( b \) can finally be calculated.

The calculation of the dot product \( \phi(x) \cdot \phi(x') \) is normally defined by a function known as the kernel function:

\[
K(x, x') = \phi(x) \cdot \phi(x')
\] (2.20)

By using different kernel functions, a wide array of nonlinear mappings \( x \rightarrow \phi(x) \) can be obtained. One popular choice of kernel function is the polynomial kernel

\[
K(x, x') = (x \cdot x' + a)^b
\] (2.21)

where \( a \) and \( b \) are user-specified parameters. An alternative is the radial basis kernel

\[
K(x, x') = e^{-\frac{|x-x'|^2}{2\sigma^2}}
\] (2.22)

where \( \sigma \) is a user-specified parameter. This parameter affects how well the SVM generalizes and must not be too small (may cause overfitting) or too large (prevents the learning of patterns). In this sense, it can be compared to the size of the MLP hidden layer.

The selection of suitable values for the parameters \( \sigma \) and \( C \) (equation 2.11) can be done by performing a grid search. This procedure consists of specifying a set of values to use for each parameter and train models on a training set with different combinations of these values. The parameters of the model which displays the best performance on unseen data are then selected as the most suitable. In order to evaluate the effect of the parameters as fairly as possible, multiple models with the same parameters should be trained and evaluated on different datasets in a cross-validation process. The \( k \)-fold cross-validation variant is explained in section 2.6.
2.6 Evaluation of machine learning algorithms

This section explains a couple of concepts related to the evaluation of the machine learning models. Section 2.6.1 describes the root mean squared error, which be used as a metric for measuring performance of regression tasks, such as lifetime prediction. Sections 2.6.2 and 2.6.3 give an insight into basic data partitioning and the $k$-fold cross-validation method, respectively.

2.6.1 Root mean squared error

The root mean squared error (RMSE) can be used to calculate the mean error over an entire dataset. For measuring the performance of a regression model, the RMSE is calculated from the predictions (numerical outputs) on the dataset used for evaluation. For a dataset with $n$ instances, the RMSE is expressed as

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

where $y_i$ represents the predicted value and $t_i$ represents the correct output value in for instance $i$. The RMSE value is measured in the same unit as $t$ and $y$.

2.6.2 Data partitioning

As previously described in section 2.4 and 2.5, data used in machine learning experiments is usually partitioned into different sets with different purposes. Most basically, data is divided into one training set used for training a model, and one test set used for evaluating the performance of a model on unseen data.

Problems arise with this simple approach if one wants to select one optimal model from a set of models (e.g. with different design parameters) and at the same time get an honest measure of the expected performance on unseen data. Would the model with the best performance on the test set be selected, this performance may not be representable for any unseen data, as the test set may have been particularly favorable for the selected model. Therefore, an additional validation set (previously described as a part of the early stopping technique in section 2.4.3) can be used instead of the test set for comparing different models. This keeps the test set reserved for evaluating the performance of the selected model and guarantees that the selected model is not biased towards the test set.

2.6.3 $k$-fold cross-validation

$k$-fold cross-validation is a method for increasing the stability of performance evaluation and is typically employed when comparing the performance of models with different design parameters. The objective is to achieve a good general view of the performance and minimize the influence of single training or validation set compositions.

This is carried out by first dividing the entire dataset into $k$ partitions. The training and validation procedure is then repeated $k$ times (folds), each using a different partition as validation set and the other $k - 1$ partitions as training set, as shown in figure 2.10 [31]. By calculating the mean of the $k$ performance measures, an overall performance measure is obtained. $k$-fold cross-validation can be used for evaluating performance of both MLP and SVM [31][32].
2.6. Evaluation of machine learning algorithms

Figure 2.10: An example of 4-fold cross-validation. Each fold uses a different validation set, while the remaining partitions make up the training set.
This chapter describes the steps of solving the lifetime prediction problem. Section 3.1 defines the considered pieces of code and how lifetime is measured. The selection of GitHub projects that were used as data sources is described in section 3.2. The process of extracting pieces of code and calculating their lifetime is explained in section 3.3. Section 3.4 describes how the different kinds of features were extracted for a piece of code. Finally, an overview of the experimental setup for lifetime prediction is given in section 3.5.

3.1 Defining lifetime prediction

As declared in chapter 1 – Introduction, the main objective of this study is to predict the lifetime for a piece of code. The lifetime will be predicted for a piece of code at the time it is added to the repository, based on the state of the Git repository and the information provided at that time. In order to do this, a piece of code must be defined as well as a metric for measuring lifetime. These definitions are given in the following sections.

3.1.1 Pieces of code

The definition of a piece of code was made with two objectives in mind. First, it should be possible to extract features that are mappable to a piece of code. This is to provide relevant information to the machine learning algorithms. Secondly, a piece of code should serve as a unit that is useful to evaluate in development work. A set of consecutively added lines of code in a Git hunk (described in section 2.1) can be deemed to meet these objectives and was thus selected to represent a piece of code in this study.

This definition of a piece of code provides more specific predictions than considering all the code changes made in an entire commit, which is the case in several examples of fault-prediction [1] [2]. This more specific approach ought to be more helpful for a developer if lifetime prediction were used in practice, as multiple unrelated code changes may have been made in the same commit. Additionally, this definition makes the predictions applicable to all additions of code and not only certain structures like classes, methods or blocks.

Figure 3.1 shows an example of how pieces of code (gray) as defined above are extracted from Git hunks. One piece of code is composed of the added lines of code (marked with “+”) in one hunk. Note that a piece of code represents lines that have been added consecutively.
3.1. Defining lifetime prediction

and it cannot be guaranteed that the lines will be kept together in the future. A piece can be “split” if new code is added between the lines of the piece without modifying them (an example is shown in section 3.1.2). Also note that the hunks in figure 3.1 contain no code surrounding the modified lines (specified by the –unified option in Git).

```java
@@ -4,2 +4,2 @@
- public class HelloWorld {
-     // Prints "Hello, Universe" to the terminal window.
-     System.out.println("Hello, Universe");
+     // Prints "Hello, World" to the terminal window.
+     System.out.println("Hello, World");
@@ -8,2 +8,2 @@
-     public int two() {
-         return 2;
+     public int one() {
+         return 1;
@@ -12 +12,3 @@
-     // A function should be here:
-     public void doNothing() {
+     // Does nothing...
+     };
```

Figure 3.1: Three pieces of code (gray) as defined in this study can be extracted from the groups of consecutive “+”-lines in each of the three Git hunks.

3.1.2 Lifetime

When defining lifetime – the numerical value to be predicted – the notion of time was based on commits as discrete timesteps rather than actual time. That is, the revision history of a Git repository was viewed as a series of commits, each representing a timestep, not taking the real time between commits into account. Hence, the unit for measuring lifetime was Git commits. The following set of rules were used to define lifetime:

- A piece of code starts to live as soon as it is added to a file.
- A piece of code is declared dead as soon as any line of it is deleted or modified. However, if new code is added within the span of a living piece (i.e. “splitting” it) without modifying the existing lines of code, the piece keeps on living.
- The lifetime of a piece of code is the number of commits between its birth and death, i.e. for how long the code has remained intact.
- Only commits that have modified the contents of the file in which the piece of code resides are counted as timesteps, i.e. changing the filename or moving the file to another folder does not count.

A few examples of how different operations affect a piece of code are given in figure 3.2.

One weakness with this definition of lifetime is that inaccuracies can occur in Git repositories with multiple branches, which have a history that contains parallel paths of commits as described in section 2.1. This may cause the lifetime of a piece of code to differ depending on which path is followed to the piece’s origin. Figure 3.3 shows an example of two branches causing the lifetime of a piece to be either 4 or 5 commits. In this study, lifetime was calculated along one path of commits. A consequence of this approach is that all changes made in parallel branches will appear as if made in a single commit when merged into the master branch, which lowers the lifetime accuracy.
3.2 Revision history dataset from GitHub

The pieces of code used for this study were extracted from five different open source repositories available on GitHub. These were selected based on the following criteria:

- **Written in Java** - Enables Java-specific approaches to feature extraction.
- **Large number of commits** - Indirectly affects the number of code pieces. As a larger training set helps learning the general patterns of the data, a large number of pieces was desired. Only repositories with at least 1000 commits were considered.
- **Popularity** - indicated by “stars” from GitHub users. This was used as an indicator of overall repository quality.

A reason for using data from more than a single repository is that behavior related to version control and code changes may be specific for a project, resulting in variations in pre-
diction performance depending on the data source. Comparing the performance on data from different sources should therefore give a better view of how precise the predictions are. Another important aspect is that a dataset composed of data from all five repositories can be used to see how well lifetime prediction works based on “global” characteristics for a piece of code (as opposed to repository-specific). Table 3.1 lists the repositories used in this study along with their number of commits and Java files.

<table>
<thead>
<tr>
<th>Repository name</th>
<th># Commits</th>
<th># Java files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrofit [33]</td>
<td>1345</td>
<td>108</td>
</tr>
<tr>
<td>Hystrix [34]</td>
<td>1830</td>
<td>263</td>
</tr>
<tr>
<td>Gitblit [35]</td>
<td>2966</td>
<td>438</td>
</tr>
<tr>
<td>Dropwizard [36]</td>
<td>3775</td>
<td>441</td>
</tr>
<tr>
<td>Okhttp [37]</td>
<td>2636</td>
<td>247</td>
</tr>
</tbody>
</table>

Table 3.1: The five GitHub repositories, their number of commits and Java files.

Purpose or functionality of the code was not used as a criterion when selecting repositories and consequently, the above repositories represent different kinds of software projects. Retrofit and Okhttp are HTTP clients for Java, Hystrix is a library for distributed systems, Gitblit is a Java solution for using Git and Dropwizard is a framework for web services.

3.3 Extraction of code pieces and lifetime

As a code pieces and lifetime were concepts defined specifically for this study, all pieces of code and their respective lifetime had to be extracted from the Git repositories. This was done by traversing the history of commits for each file in a repository while keeping track of code pieces getting added and dying. All pieces that were alive at a step of the traversal had their lifetime counter increased by one. As soon as a piece of code died, its counter was no longer increased. When the traversal was done, records of the dead pieces (for which the lifetime had a known value) were output.

Algorithm 1 describes the procedure in more detail. Two sets are used for keeping track of code pieces: living_pieces and dead_pieces. In these sets, a code piece is represented by a code piece record, which contains a lifetime counter and a data structure for storing feature values (features are discussed in section 3.4). The sets store the records by a unique id. An example set containing two code piece records with ids 2 and 45 looks as follows:

\[
\begin{align*}
2 & : \{\text{counter} : 8, \text{features} : [\text{feature values}]\}, \\
45 & : \{\text{counter} : 1, \text{features} : [\text{feature values}]\}
\end{align*}
\]

In algorithm 1, living_pieces and dead_pieces are initialized to empty sets (lines 1-2). The algorithm then iterates over all the files in the repository (line 3). For each file, the algorithm iterates over the file’s history of commits (line 4). Note that this iteration only spans from the second commit to the last, with the first commit omitted. This was done because the first commit is likely to consist of one – potentially large – piece of code which will be modified in the second commit, giving it a lifetime of 1. This was considered to be a special case, non-representative of how code is usually added in a commit. The last commit is equivalent to the most recent commit at the time of the procedure.

For each commit, two sets of ids are generated: new_ids, which contains new unique ids for the pieces of code that have been added in the commit (line 5) and dead_ids, which contains the ids of pieces that have died in the commit (line 6). The procedures for determining which pieces have been added or died in a commit are explained later on.

The pieces that have died in the commit have their code piece records moved from living_pieces to dead_pieces after being identified using dead_ids (lines 7-9). For each of the
new ids in new_ids, a code piece record is created and added to living_pieces (lines 10-12). After this, the lifetime counter is incremented for each of the living pieces of code (lines 13-15).

Finally, dead_pieces is returned, while living_pieces is not. This is because the living pieces of code have no definite lifetime value and there is no way of determining it since they may or may not die at some unknown point in the future. It is known, however, that the living pieces have a lifetime that is larger than a certain value, but due to the design of this study, they were truncated. The dead pieces of code comprised about half of the pieces in each of the five repositories and a slight majority in total. This is shown in chapter 4 – Results.

Algorithm 1 Pseudocode describing the procedure of extracting code pieces and their lifetime.

1: living_pieces ← []
2: dead_pieces ← []
3: for all file ∈ repository do
4:    for commit c := second to last do
5:        new_ids ← ids of pieces added in c
6:        dead_ids ← ids of pieces having died in c
7:        for all piece_id ∈ dead_ids do
8:            Move code piece record with id == piece_id from living_pieces to dead_pieces
9:        end for
10:       for all piece_id ∈ new_ids do
11:           Add a code piece record with id == piece_id and counter == 0 to living_pieces
12:       end for
13:       for all code piece record ∈ living_pieces do
14:           Increase its counter by 1
15:       end for
16:    end for
17: end for
18: return dead_pieces

Two crucial parts of algorithm 1 which are not explained in the pseudocode are the tasks of determining which pieces have been added or died, respectively, in each commit. The first task was solved by using the Git diff command to compare the current and previous version of the currently processed file. As the consecutively added lines of code represent a piece of code, one id per hunk (given that the hunk contained an addition of code) could simply be created from this information. The new ids were generated from an incrementing counter.

For determining if a piece of code has died in the current commit, it is first necessary to keep track of which lines in the file a piece spans over. Therefore, an additional data structure, called lines_array was used. lines_array is a resizable array, with a size that corresponds to the number of lines in the currently processed file plus one. The basic idea is that an element at index i of lines_array is a number that corresponds to the id of the piece that line i in the currently processed file belongs to (i.e. the piece in which the line was added). By keeping lines_array updated with the changes (lines added/deleted) in every commit, it can simply be checked which piece of code a specific line in the file belongs to.

The information needed for updating lines_array is provided by the hunks obtained from the Git diff command (with the –unified option set to exclude unmodified code from the hunks). As each hunk contains line numbers of added and deleted lines, these can be used to delete and add elements to lines_array. These operations need to be performed with caution in order to keep lines_array consistent, meaning that elements must be deleted in reverse order (i.e. highest index first) in order for the indices to stay correct. Elements were added with lowest indices first for the same reason. Another important detail is that in the case of multiple changes made in a single commit (resulting in multiple hunks in the diff), the delete operations of all these changes were performed before adding any elements.
Having the `lines_array` array up to date, it could easily be checked by inspecting every element of `lines_array` that was about to be deleted if a piece of code had died in the current commit. If the value of this element corresponded to an id of a code piece record in `living_pieces`, the record was moved to `dead_pieces`.

A simplified overview of the explained procedures is given below in algorithms 2 and 3. `living_pieces` and `lines_array` can be considered as global variables.

**Algorithm 2** Remove elements from `lines_array` and collect ids of all dead pieces.

1. `dead_ids ← {}`
2. `hunks ← hunks in Git diff`
3. `minus ← {}`
4. `for all h ∈ hunks do`
5.  `add line numbers of all deleted lines to minus`
6. `end for`
7. `for all m ∈ descending_order(minus) do`
8.  `if ∃ code piece record ∈ living_pieces with id == lines_array[m] then`
9.   `add id to dead_ids`
10. `end if`
11. `lines_array.remove(m) {remove element at index m}`
12. `end for`
13. `return dead_ids`

**Algorithm 3** Add elements to `lines_array` and generate ids for new code pieces.

1. `new_ids ← {}`
2. `hunks ← hunks in Git diff`
3. `piece_intervals ← {}`
4. `for all h ∈ hunks do`
5.  `add interval of consecutively added lines to piece_intervals`
6. `end for`
7. `for all p ∈ piece_intervals do`
8.  `id ← a new unique id`
9. `add id to new_ids`
10. `for i = p.start to p.end do`
11.  `lines_array.insert(i, id) {insert id at index i}`
12. `end for`
13. `end for`
14. `return new_ids`

### 3.4 Feature extraction

In this study, the three different feature groups mentioned in chapter 2—Theory were used for describing a piece of code:

- Code contents
- Static code analysis
- Change metadata

The features were all extracted from either Git or file contents with respect to the repository state at the time when the piece of code was added. This is because the predictions – if applied in practice – would likely be of most interest when applied to newly written code, i.e. code that has just been added to the repository.
3.4. Feature extraction

The extraction procedure was combined with the Git history traversal described in the previous section and the features of a piece of code were stored in its code piece record. In the following sections the feature groups are specified more closely as well as the extraction process of the features.

3.4.1 Code contents

A bag-of-words model was used for extracting features from the contents of the code. The feature values represented term frequency. Regarding which terms to include in the vocabulary, two different approaches were taken:

- **Reserved terms only** - Java keywords, separators, operators and special characters.
- **All terms** - All terms that can be found in the code.

The idea of extracting reserved terms only is that identifiers (e.g. variable names) that may be very project-specific are avoided, thus leading to a more general solution. Only terms that are specified as lexical elements in the Java language specification [38] were used (105 in total).

The second approach extracts all terms (including those in the first approach). An obvious disadvantage of using all terms in the code is that the vector grows very large compared to the first approach (from the five repositories, a total of 11336 code terms were extracted). Therefore, the most frequent terms were filtered out to be used as features. In this study, two different levels of filtering were used: the 1 % and 10 % most frequent terms.

In order to make the code more general, all identifiers written in camelCase (customary for Java [39]) were split to separate words and cast to lower case letters. For example:

- `addWriterModule` → `{add, writer, module}
- `addReaderModule` → `{add, reader, module}

This makes it possible for two identifiers to be similar without having to be identical, which may help the machine learning algorithms in recognizing patterns.

The code contents were extracted from the information given by the Git `diff` command by filtering out the terms from the lines added in a hunk (the lines marked “+”). The extracted code terms were stored along with their respective number of occurrences in the code piece records. A vocabulary, on which filtering could be applied, was constructed after all the code pieces in the dataset had been extracted. The filtered vocabulary was then used to select terms from the code piece records when transforming them into feature vectors.

3.4.2 Static code analysis

A group of features was extracted using the static code analysis tool PMD [40]. PMD takes as input a source code file and outputs a list of warnings and metrics related to the code.

PMD is suitable for extracting features since line numbers are provided along with the warning or calculated metric, which is useful for mapping the information to code pieces. This may not make it possible to map the output to a code piece specifically, but it helps determining if a piece overlaps with code that has certain properties. In this study, overlap was used as an indicator of whether or not to map PMD output to a piece of code. As an example, consider the piece of code in figure 3.4 which spans over lines 5-12 (gray). In this example, the code in the figure has been analyzed with PMD, resulting in the following output:

- One warning related to the entire `Square` class (lines 1-19). This overlaps with the code piece and is therefore used as a feature to describe it.
3.4. Feature extraction

- One complexity metric related to the `Square` constructor (lines 7-12). This also overlaps with the code piece and is therefore used as a feature to describe it.

- One warning related to the `getRot()` function (lines 14-18). This does not overlap with the code piece and is not used as a feature, since it is considered irrelevant.

```java
public abstract class Square{
    public boolean north, east, south, west;
    protected int rotation = 0;

    private SquareGenerator myGenerator = new SquareGenerator();

    public Square(boolean north, boolean east, boolean south, boolean west) {
        this.north = north;
        this.east = east;
        this.south = south;
        this.west = west;
    }

    public abstract int getType();

    public int getRot() {
        return rotation;
    }
}
```

Figure 3.4: A piece of code (gray) plus some surrounding code.

In order to obtain as much information as possible from the static code analysis, all available PMD rulesets (specifying what warnings and metrics to generate) for Java were used. In total, 195 features were extracted using PMD, most of which were warnings of potential errors, violations of code standards and possibly suboptimal code. 7 of the features were metrics related to code complexity, such as cyclomatic complexity. These were obtained by setting minimal threshold values of the rules in the `codesize` ruleset. Aside from that, only default settings were used.

The features derived from warnings were represented by the number of occurrences within the code piece (if the piece is large, the same warnings may occur multiple times) and the metrics by their numerical value.

As previously mentioned, PMD takes a source code file as input. This means that in order for features to be extracted from PMD, the input file must be same the version of the file as in the commit when the code piece was added. Using Git, this was done by checking out every commit when traversing the history of a file, which reverts the file to its historical version.

3.4.3 Change metadata

The set of features related to change metadata was based on a combination of previous studies and features that were easily obtainable from Git. The following features were used:

- Developer experience: The number of previous commits by the committing developer that include the specific file.

- Developer count: The number of developers that have made commits to the file.

- File change count: The number of commits that have affected the file.
3.5 Experiments

- Lines of code (LOC) in file
- LOC in code piece
- Log message length: The number of character in the log message.
- Log message terms: Represented by a bag-of-words model.

Developer experience, developer count, and file change count were obtained with the Git log command combined with the -follow and -pretty options. The code piece LOC was derived from the hunk line numbers in the extraction of code pieces (section 3.3) and file LOC from counting the lines in the file. For sake of the static code analysis in section 3.4.2 all files were reverted to previous versions in the history traversal, which made this particular LOC solution possible.

Similarly to the code contents, the number of occurrences of each log message term was stored in the code piece record and a vocabulary was constructed after all pieces had been extracted. Filtering was applied on the vocabulary by selecting the most frequent terms. In this study, the 1 % and 10 % most common terms were used. To measure the influence of using log message terms at all, 0 % were used for experiments as well. In total, 5184 log messages terms were extracted from the five repositories.

Note that the majority of these features cannot be mapped to a specific piece of code, but a specific commit. Pieces added in the same commit will therefore have largely the same values of the change metadata features, which motivates the use of other (piece specific) features as well.

3.5 Experiments

This section explains the experimental setup of the study. Section 3.5.1 provides an overview of the experiments performed for evaluating lifetime predictions. Sections 3.5.2 and 3.5.3 describe how the design parameters were selected for the MLP and SVM models.

3.5.1 Overview

Before the experiments, the feature information stored in the code piece records was transformed into feature vectors. All features were standardized, giving the values a mean of 0 and a standard deviation of 1. This was done in order to aid the machine learning algorithms since the original feature values were all positive and in very different ranges.

In order to explore the usefulness of different features, lifetime prediction experiments were performed with different combinations of feature groups representing the code pieces. Usefulness was approximated by observing the changes in performance (RMSE) on the test set when including and excluding certain feature groups. While this suffices for approximation, it is not enough for telling the exact influence on the result of individual feature groups in a combination. That is, the importance of a feature may differ depending on which features it is combined with. In this study, the following feature groups were used:

- \( C_1 \) – Code contents (1 % most frequent terms)
- \( C_{10} \) – Code contents (10 % most frequent terms)
- \( C_r \) – Code contents (reserved terms only)
- \( S \) – Static code analysis
- \( M_0 \) – Change metadata (0 % log message terms)
- \( M_1 \) – Change metadata (1 % most frequent log message terms)
3.5. Experiments

- $M_{10}$ – Change metadata (10 % most frequent log message terms)

In addition to using the above feature groups separately, combinations (unions) of multiple feature groups were used. In these combinations, the $C_1$ variant of frequent code terms was excluded in order to reduce the number of experiments. For the same reason, the $M_0$ and $M_1$ variants of change metadata were excluded as well. Besides, the 10 % filtering was deemed likely to perform best and would consequently be of most interest when combining different feature groups. However, for code contents, $C_r$ (reserved Java terms only) was included in combinations as well. This resulted in the following combinations being used:

- $C_{10}S$
- $C_rS$
- $C_{10}M_{10}$
- $C_rM_{10}$
- $SM_{10}$
- $C_{10}SM_{10}$
- $C_rSM_{10}$

For each of the above sets of features (both single groups and combinations), six MLP and SVM models were produced: five which were trained and tested on each of the Git repositories separately and one which was trained and tested on data from all five repositories. This was done to see if the characteristics that influence lifetime are common between different data sources and how training on more diverse data affects the results.

The design parameters of the models used for the final performance evaluation were selected in a 4-fold cross-validation process using 80 % of the datasets for training and validation. The remaining 20 % was used only as a test set for the final evaluation. Sections 3.5.2 and 3.5.3 provide details on how design parameters were selected for MLP and SVM, respectively. Figure 3.5 shows an overview of the experimental steps described above.

For each dataset and set of features:

**Divide dataset:**

- Training/Validation
- Test

**Select design parameters:**

- For both MLP and SVM:
  - Perform 4-fold cross-validation with different design parameters.
  - Select the design parameters that yield the smallest average RMSE.
  - Train one final model with these parameters.

**Evaluate performance on new data:**

- For final MLP and SVM models:
  - Obtain the RMSE for predictions on test data.

Figure 3.5: Overview of the experimental steps.
3.5. Experiments

Note that predictions made for evaluating performance correspond to predicting the lifetime of each piece of code in the test set as if it was just added to a Git repository. This is because the features that describe each piece of code were extracted from data that was available at the point in time it was added.

3.5.2 Selecting MLP parameters

The size of the hidden layer was determined by measuring the prediction performance on a validation set with different layer sizes and selecting the size which yielded the smallest RMSE. 4-fold cross-validation was used in order to increase the stability of the results.

For each combination of feature groups, data from one single Git repository (Hystrix) was used to select a ratio between the hidden layer size and the input layer size. This ratio was then reused when training models on the remaining five datasets for the same combination of feature groups. The reason for this was to save time, as network training is a time-consuming process. The decision was also based on the fact that the appropriate size of the hidden layer depends on the degree of nonlinearity of the problem at hand [12] and this was assumed to be roughly similar for all datasets.

The multilayer perceptron used in the experiments was composed of three layers: input, output and one hidden layer. The hyperbolic tangent was selected as activation function in the hidden layer. Combined with the standardization of the features, this can make training more effective [16]. Since an unbounded numerical output was desired, a linear activation function was used for the output neuron.

The network was trained with the backpropagation algorithm described in section 2.4.2 in an online learning process, updating the weights after each input. The error on the validation set was calculated every fifth epoch and training was aborted when the error had increased for three consecutive observations. In addition to the early stopping criterion, a maximum number of epochs was set to 2000 and a lower limit of the training set error was set to 0.1. However, none of these limits were reached in any experiment.

Based on the heuristics listed in section 2.4.4, six different hidden layer sizes were tested: 0.47, 0.57, 0.67, 0.77, 0.87 and 0.97 times the input layer size. The learning rate was set to 10^{-4} and the momentum to 0.5 in all experiments.

3.5.3 Selecting SVM parameters

For the SVM experiments, a radial basis function (RBF) kernel was used, due to its ability to handle nonlinear cases and the fact that is has fewer parameters to set than a polynomial kernel, which makes the process of selecting a model less complex [15]. Similarly to MLP, the design parameters were selected using data from one single repository (Hystrix) and reused for all models that were trained with the same feature group combination.

The values of $\sigma$ (used in the kernel function) and $C$ (used for penalizing data points outside the $\epsilon$ tube) were determined by measuring the prediction performance on the validation set with different combinations in a grid search [15]. 4-fold cross-validation was used in order to increase the stability of the results. The tested values of $C$ were $2^{-2}, 2^0, \ldots, 2^{10}$ and of $\sigma$ $2^{-10}, 2^{-8}, \ldots, 2^0$. The parameter $\epsilon$ was kept constant at 0.1.
This chapter presents the results obtained from the different steps in the study. Section 4.1 concerns the extracted pieces of code from each repository and their lifetime. Section 4.2 presents the number of extracted features from each repository. Sections 4.3 and 4.4 show the performance of lifetime prediction on the Hystrix repository with different parameter settings of the MLP and SVM algorithms. An overview of the results from the entire series of lifetime prediction experiments is given in section 4.5.

4.1 Lifetime extraction

Table 4.1 displays the number of code pieces obtained from the extraction process described in section 3.3. The percentage of dead pieces is also displayed in order to show the distribution of dead and living pieces in each repository. Only the dead pieces were used as data for this study.

<table>
<thead>
<tr>
<th>Repository</th>
<th># Dead pieces</th>
<th># Living pieces</th>
<th>% Dead pieces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrofit</td>
<td>4686</td>
<td>2525</td>
<td>64.9</td>
</tr>
<tr>
<td>Hystrix</td>
<td>5457</td>
<td>5539</td>
<td>49.6</td>
</tr>
<tr>
<td>Gitblit</td>
<td>13478</td>
<td>14511</td>
<td>48.2</td>
</tr>
<tr>
<td>Dropwizard</td>
<td>5853</td>
<td>6051</td>
<td>49.2</td>
</tr>
<tr>
<td>Okhttp</td>
<td>14652</td>
<td>10022</td>
<td>59.4</td>
</tr>
<tr>
<td>All repositories</td>
<td>44126</td>
<td>38648</td>
<td>53.3</td>
</tr>
</tbody>
</table>

Table 4.1: The number of extracted code pieces and the percentage of dead pieces.

The pieces of code in different repositories had different lifetime properties in terms of mean and standard deviation. These measures for each repository are displayed in table 4.2. The standard deviation describes how large the variations of lifetime in a dataset are, i.e. the bigger difference between the values in the dataset and the mean, the larger the standard deviation. Lifetime is measured in Git commits, as explained in section 3.1.2.

The empirical CDF (cumulative distribution function) plot in figure 4.1 further shows the lifetime distribution in the different repositories. The Y-axis represents the cumulative percentage of code pieces with a certain lifetime value (X-axis).
4.2 Feature extraction

Table 4.2: Mean lifetime and standard deviation of the different repositories.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Mean lifetime (commits)</th>
<th>Standard deviation (commits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrofit</td>
<td>8.79</td>
<td>12.82</td>
</tr>
<tr>
<td>Hystrix</td>
<td>9.16</td>
<td>11.73</td>
</tr>
<tr>
<td>Gitblit</td>
<td>16.37</td>
<td>31.40</td>
</tr>
<tr>
<td>Dropwizard</td>
<td>4.78</td>
<td>5.14</td>
</tr>
<tr>
<td>Okhttp</td>
<td>10.42</td>
<td>14.69</td>
</tr>
<tr>
<td>All repositories</td>
<td>11.16</td>
<td>20.64</td>
</tr>
</tbody>
</table>

Figure 4.1: The empirical CDF of lifetime for the different repositories.

4.2 Feature extraction

Table 4.3 shows the number of features extracted from the repositories. The number of features varies between different repositories since it depends on the output generated by PMD and the diversity of terms used in the code and log messages. The features listed for All repositories represents the union of the features extracted from all five repositories. All the numbers represent the number of features before any filtering is applied.

Seven of the features in the static code analysis group are code complexity metrics which are generated for all pieces of code and independent of warnings generated by PMD. Similarly, the change metadata group contains six metrics that are extracted from Git and unrelated to terms in log messages and thus always generated for a piece of code. Appendix A provides a summary of the most frequently occurring code terms, PMD warnings and log message terms in each dataset.
4.3 Setting the hidden layer for MLP

An appropriate size of the hidden layer was determined by measuring the lifetime prediction performance with different layer sizes on data from the Hystrix repository. Performance was measured by the average RMSE in a 4-fold cross-validation process, using 80% of the original dataset.

One layer size was selected for each set of features, i.e., individual feature groups as well as combinations of several feature groups. Six different sizes were tested, based on the ratio between the hidden layer size and input layer size: 0.47, 0.57, 0.67, 0.77, 0.87, and 0.97.

Table 4.4 displays the average RMSE for each layer size with data represented by single feature groups and table 4.5 displays the average RMSE for combinations of feature groups. The best result obtained for each set of features is displayed in gray, and corresponds to the size that was selected for the final models.

**Table 4.3: The number of features in each group extracted from the repositories.**

<table>
<thead>
<tr>
<th>Repository</th>
<th>Code contents</th>
<th>Static code analysis</th>
<th>Change metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrofit</td>
<td>1867</td>
<td>116</td>
<td>950</td>
</tr>
<tr>
<td>Hystrix</td>
<td>2980</td>
<td>127</td>
<td>1337</td>
</tr>
<tr>
<td>Gitbl</td>
<td>4903</td>
<td>158</td>
<td>2594</td>
</tr>
<tr>
<td>Dropwizard</td>
<td>3295</td>
<td>119</td>
<td>1840</td>
</tr>
<tr>
<td>Okhttp</td>
<td>6420</td>
<td>144</td>
<td>1742</td>
</tr>
<tr>
<td>All repositories</td>
<td>11336</td>
<td>195</td>
<td>5190</td>
</tr>
</tbody>
</table>

**Table 4.4: Average RMSE on validation sets, obtained with different hidden layer sizes and single feature groups.**

<table>
<thead>
<tr>
<th>Hidden/input layer ratio</th>
<th>RMSE (commits) for individual feature groups</th>
<th>C1</th>
<th>C10</th>
<th>C7</th>
<th>S</th>
<th>M0</th>
<th>M1</th>
<th>M10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.47</td>
<td></td>
<td>11.14</td>
<td>9.82</td>
<td>10.82</td>
<td>8.63</td>
<td>9.43</td>
<td>8.35</td>
<td>8.05</td>
</tr>
<tr>
<td>0.57</td>
<td></td>
<td>11.22</td>
<td>9.77</td>
<td>10.85</td>
<td>8.59</td>
<td>8.99</td>
<td>8.15</td>
<td>8.06</td>
</tr>
<tr>
<td>0.67</td>
<td></td>
<td>11.17</td>
<td>9.79</td>
<td>10.83</td>
<td>8.73</td>
<td>8.88</td>
<td>8.12</td>
<td>8.08</td>
</tr>
<tr>
<td>0.77</td>
<td></td>
<td>11.15</td>
<td>9.84</td>
<td>10.82</td>
<td>8.64</td>
<td>8.66</td>
<td>7.98</td>
<td>8.09</td>
</tr>
<tr>
<td>0.87</td>
<td></td>
<td>11.25</td>
<td>9.90</td>
<td>10.79</td>
<td>8.68</td>
<td>8.65</td>
<td>7.94</td>
<td>8.10</td>
</tr>
<tr>
<td>0.97</td>
<td></td>
<td>11.17</td>
<td>9.86</td>
<td>10.87</td>
<td>8.66</td>
<td>8.32</td>
<td>8.00</td>
<td>8.09</td>
</tr>
</tbody>
</table>

**Table 4.5: Average RMSE on validation sets, obtained with different hidden layer sizes and feature group combinations.**

<table>
<thead>
<tr>
<th>Hidden/input layer ratio</th>
<th>RMSE (commits) for feature group combinations</th>
<th>C10S</th>
<th>C7S</th>
<th>C10M10</th>
<th>C7M10</th>
<th>SM10</th>
<th>C10SM10</th>
<th>C7SM10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.47</td>
<td></td>
<td>8.24</td>
<td>8.56</td>
<td>7.30</td>
<td>8.09</td>
<td>7.18</td>
<td>7.20</td>
<td>7.33</td>
</tr>
<tr>
<td>0.57</td>
<td></td>
<td>8.36</td>
<td>8.56</td>
<td>7.28</td>
<td>8.06</td>
<td>7.17</td>
<td>7.22</td>
<td>7.28</td>
</tr>
<tr>
<td>0.67</td>
<td></td>
<td>8.28</td>
<td>8.59</td>
<td>7.29</td>
<td>8.07</td>
<td>7.10</td>
<td>7.22</td>
<td>7.31</td>
</tr>
<tr>
<td>0.77</td>
<td></td>
<td>8.29</td>
<td>8.48</td>
<td>7.29</td>
<td>7.96</td>
<td>7.16</td>
<td>7.16</td>
<td>7.29</td>
</tr>
<tr>
<td>0.87</td>
<td></td>
<td>8.36</td>
<td>8.59</td>
<td>7.33</td>
<td>8.07</td>
<td>7.20</td>
<td>7.09</td>
<td>7.30</td>
</tr>
<tr>
<td>0.97</td>
<td></td>
<td>8.29</td>
<td>8.59</td>
<td>7.38</td>
<td>8.13</td>
<td>7.11</td>
<td>7.13</td>
<td>7.30</td>
</tr>
</tbody>
</table>
4.4 Setting the parameters for SVM

The C and \( \sigma \) parameters were selected by measuring the prediction performance with different combinations in a grid search. Data from the Hystrix repository was used, same as for determining the MLP hidden layer size. Performance was measured by the average RMSE in a 4-fold cross-validation process, using 80% of the original dataset.

Tables 4.6-4.19 display the average RMSE on the validation sets for each combination of C and \( \sigma \) for the different sets of features. The best result obtained for each set of features is displayed in gray, and corresponds to the combination of C and \( \sigma \) selected for the final models.

### Table 4.6: Grid search for \( C_1 \)

<table>
<thead>
<tr>
<th>C</th>
<th>( \sigma )</th>
<th>( 2^{-10} )</th>
<th>( 2^{-8} )</th>
<th>( 2^{-6} )</th>
<th>( 2^{-4} )</th>
<th>( 2^{-2} )</th>
<th>( 2^0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2^{-2} )</td>
<td>12.75</td>
<td>12.72</td>
<td>12.65</td>
<td>12.52</td>
<td>12.29</td>
<td>12.06</td>
<td></td>
</tr>
<tr>
<td>( 2^{-4} )</td>
<td>12.20</td>
<td>12.63</td>
<td>12.50</td>
<td>12.21</td>
<td>11.84</td>
<td>11.54</td>
<td></td>
</tr>
<tr>
<td>( 2^0 )</td>
<td>12.63</td>
<td>12.52</td>
<td>12.25</td>
<td>11.84</td>
<td>11.46</td>
<td>11.17</td>
<td></td>
</tr>
<tr>
<td>( 2^2 )</td>
<td>12.54</td>
<td>12.35</td>
<td>11.99</td>
<td>11.59</td>
<td>11.27</td>
<td>10.84</td>
<td></td>
</tr>
<tr>
<td>( 2^4 )</td>
<td>12.45</td>
<td>12.17</td>
<td>11.77</td>
<td>11.43</td>
<td>11.19</td>
<td>10.43</td>
<td></td>
</tr>
<tr>
<td>( 2^6 )</td>
<td>12.39</td>
<td>11.98</td>
<td>11.68</td>
<td>11.36</td>
<td>11.06</td>
<td>10.37</td>
<td></td>
</tr>
<tr>
<td>( 2^{10} )</td>
<td>12.38</td>
<td>11.93</td>
<td>11.70</td>
<td>11.44</td>
<td>10.98</td>
<td>10.88</td>
<td></td>
</tr>
</tbody>
</table>

### Table 4.7: Grid search for \( C_{10} \)

<table>
<thead>
<tr>
<th>C</th>
<th>( \sigma )</th>
<th>( 2^{-10} )</th>
<th>( 2^{-8} )</th>
<th>( 2^{-6} )</th>
<th>( 2^{-4} )</th>
<th>( 2^{-2} )</th>
<th>( 2^0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2^{-2} )</td>
<td>12.51</td>
<td>12.22</td>
<td>12.00</td>
<td>11.97</td>
<td>12.05</td>
<td>12.21</td>
<td></td>
</tr>
<tr>
<td>( 2^{-4} )</td>
<td>12.19</td>
<td>11.79</td>
<td>11.39</td>
<td>11.07</td>
<td>11.11</td>
<td>11.39</td>
<td></td>
</tr>
<tr>
<td>( 2^0 )</td>
<td>11.83</td>
<td>11.27</td>
<td>10.56</td>
<td>9.99</td>
<td>10.05</td>
<td>10.29</td>
<td></td>
</tr>
<tr>
<td>( 2^2 )</td>
<td>11.49</td>
<td>10.79</td>
<td>9.94</td>
<td>9.56</td>
<td>9.37</td>
<td>9.53</td>
<td></td>
</tr>
<tr>
<td>( 2^4 )</td>
<td>11.14</td>
<td>10.36</td>
<td>9.64</td>
<td>9.36</td>
<td>9.36</td>
<td>9.56</td>
<td></td>
</tr>
<tr>
<td>( 2^6 )</td>
<td>10.94</td>
<td>9.96</td>
<td>9.65</td>
<td>9.38</td>
<td>9.43</td>
<td>9.58</td>
<td></td>
</tr>
<tr>
<td>( 2^{10} )</td>
<td>10.78</td>
<td>10.18</td>
<td>9.87</td>
<td>9.63</td>
<td>9.52</td>
<td>9.58</td>
<td></td>
</tr>
</tbody>
</table>

### Table 4.8: Grid search for \( C_r \)

<table>
<thead>
<tr>
<th>C</th>
<th>( \sigma )</th>
<th>( 2^{-10} )</th>
<th>( 2^{-8} )</th>
<th>( 2^{-6} )</th>
<th>( 2^{-4} )</th>
<th>( 2^{-2} )</th>
<th>( 2^0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2^{-2} )</td>
<td>12.58</td>
<td>12.5</td>
<td>12.28</td>
<td>12.03</td>
<td>11.95</td>
<td>11.86</td>
<td></td>
</tr>
<tr>
<td>( 2^{-4} )</td>
<td>12.51</td>
<td>12.28</td>
<td>12.06</td>
<td>11.86</td>
<td>11.67</td>
<td>11.51</td>
<td></td>
</tr>
<tr>
<td>( 2^0 )</td>
<td>12.33</td>
<td>12.10</td>
<td>11.81</td>
<td>11.59</td>
<td>11.29</td>
<td>11.06</td>
<td></td>
</tr>
<tr>
<td>( 2^2 )</td>
<td>12.18</td>
<td>11.91</td>
<td>11.65</td>
<td>11.34</td>
<td>10.91</td>
<td>10.81</td>
<td></td>
</tr>
<tr>
<td>( 2^4 )</td>
<td>12.10</td>
<td>11.78</td>
<td>11.52</td>
<td>11.17</td>
<td>10.83</td>
<td>10.77</td>
<td></td>
</tr>
<tr>
<td>( 2^6 )</td>
<td>12.01</td>
<td>11.73</td>
<td>11.44</td>
<td>11.05</td>
<td>10.85</td>
<td>10.77</td>
<td></td>
</tr>
<tr>
<td>( 2^{10} )</td>
<td>12.04</td>
<td>11.74</td>
<td>11.56</td>
<td>11.24</td>
<td>11.05</td>
<td>10.82</td>
<td></td>
</tr>
</tbody>
</table>
### 4.4. Setting the parameters for SVM

#### Table 4.9: Grid search for $S$

<table>
<thead>
<tr>
<th>$\sigma$</th>
<th>$2^{-10}$</th>
<th>$2^{-8}$</th>
<th>$2^{-6}$</th>
<th>$2^{-4}$</th>
<th>$2^{-2}$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-2}$</td>
<td>11.63</td>
<td>11.15</td>
<td>10.92</td>
<td>10.76</td>
<td>11.01</td>
<td>11.38</td>
</tr>
<tr>
<td>$2^0$</td>
<td>11.11</td>
<td>10.79</td>
<td>10.40</td>
<td>10.15</td>
<td>10.09</td>
<td>10.23</td>
</tr>
<tr>
<td>$2^2$</td>
<td>10.79</td>
<td>10.43</td>
<td>10.03</td>
<td>9.43</td>
<td>9.22</td>
<td>9.24</td>
</tr>
<tr>
<td>$2^4$</td>
<td>10.53</td>
<td>10.19</td>
<td>9.57</td>
<td>8.86</td>
<td>8.56</td>
<td>8.52</td>
</tr>
<tr>
<td>$2^8$</td>
<td>10.20</td>
<td>9.62</td>
<td>8.93</td>
<td>8.50</td>
<td>8.39</td>
<td>8.53</td>
</tr>
<tr>
<td>$2^{-10}$</td>
<td>10.01</td>
<td>9.37</td>
<td>8.92</td>
<td>8.62</td>
<td>8.62</td>
<td>8.87</td>
</tr>
</tbody>
</table>

#### Table 4.10: Grid search for $M_0$

<table>
<thead>
<tr>
<th>$\sigma$</th>
<th>$2^{-10}$</th>
<th>$2^{-8}$</th>
<th>$2^{-6}$</th>
<th>$2^{-4}$</th>
<th>$2^{-2}$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-2}$</td>
<td>12.41</td>
<td>11.77</td>
<td>11.19</td>
<td>10.46</td>
<td>9.83</td>
<td>9.42</td>
</tr>
<tr>
<td>$2^0$</td>
<td>11.77</td>
<td>11.20</td>
<td>10.54</td>
<td>9.94</td>
<td>9.26</td>
<td>8.39</td>
</tr>
<tr>
<td>$2^2$</td>
<td>11.20</td>
<td>10.66</td>
<td>10.16</td>
<td>9.54</td>
<td>8.86</td>
<td>8.11</td>
</tr>
<tr>
<td>$2^4$</td>
<td>10.71</td>
<td>10.38</td>
<td>9.79</td>
<td>9.36</td>
<td>8.45</td>
<td>8.26</td>
</tr>
<tr>
<td>$2^6$</td>
<td>10.54</td>
<td>10.08</td>
<td>9.60</td>
<td>9.13</td>
<td>8.23</td>
<td>8.40</td>
</tr>
<tr>
<td>$2^8$</td>
<td>10.35</td>
<td>9.81</td>
<td>9.41</td>
<td>8.93</td>
<td>8.24</td>
<td>8.47</td>
</tr>
<tr>
<td>$2^{-10}$</td>
<td>10.07</td>
<td>9.64</td>
<td>9.28</td>
<td>8.73</td>
<td>8.51</td>
<td>8.52</td>
</tr>
</tbody>
</table>

#### Table 4.11: Grid search for $M_1$

<table>
<thead>
<tr>
<th>$\sigma$</th>
<th>$2^{-10}$</th>
<th>$2^{-8}$</th>
<th>$2^{-6}$</th>
<th>$2^{-4}$</th>
<th>$2^{-2}$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-2}$</td>
<td>12.11</td>
<td>11.70</td>
<td>10.91</td>
<td>10.22</td>
<td>10.22</td>
<td>10.53</td>
</tr>
<tr>
<td>$2^0$</td>
<td>11.67</td>
<td>10.94</td>
<td>10.08</td>
<td>9.29</td>
<td>8.86</td>
<td>8.76</td>
</tr>
<tr>
<td>$2^2$</td>
<td>10.99</td>
<td>10.35</td>
<td>9.44</td>
<td>8.71</td>
<td>8.28</td>
<td>8.23</td>
</tr>
<tr>
<td>$2^4$</td>
<td>10.51</td>
<td>9.73</td>
<td>9.12</td>
<td>8.30</td>
<td>8.07</td>
<td>8.01</td>
</tr>
<tr>
<td>$2^6$</td>
<td>10.11</td>
<td>9.41</td>
<td>8.66</td>
<td>8.22</td>
<td>8.04</td>
<td>8.07</td>
</tr>
<tr>
<td>$2^8$</td>
<td>9.66</td>
<td>9.20</td>
<td>8.36</td>
<td>8.23</td>
<td>8.24</td>
<td>8.29</td>
</tr>
<tr>
<td>$2^{-10}$</td>
<td>9.41</td>
<td>8.80</td>
<td>8.35</td>
<td>8.28</td>
<td>8.45</td>
<td>8.56</td>
</tr>
</tbody>
</table>

#### Table 4.12: Grid search for $M_{10}$

<table>
<thead>
<tr>
<th>$\sigma$</th>
<th>$2^{-10}$</th>
<th>$2^{-8}$</th>
<th>$2^{-6}$</th>
<th>$2^{-4}$</th>
<th>$2^{-2}$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-2}$</td>
<td>11.91</td>
<td>11.27</td>
<td>10.97</td>
<td>10.93</td>
<td>10.92</td>
<td>10.98</td>
</tr>
<tr>
<td>$2^2$</td>
<td>10.40</td>
<td>9.47</td>
<td>8.97</td>
<td>8.57</td>
<td>8.33</td>
<td>8.29</td>
</tr>
<tr>
<td>$2^4$</td>
<td>9.73</td>
<td>9.00</td>
<td>8.51</td>
<td>8.29</td>
<td>8.14</td>
<td>8.16</td>
</tr>
<tr>
<td>$2^6$</td>
<td>9.25</td>
<td>8.70</td>
<td>8.38</td>
<td>8.27</td>
<td>8.24</td>
<td>8.31</td>
</tr>
<tr>
<td>$2^8$</td>
<td>8.98</td>
<td>8.48</td>
<td>8.40</td>
<td>8.37</td>
<td>8.34</td>
<td>8.27</td>
</tr>
<tr>
<td>$2^{-10}$</td>
<td>8.70</td>
<td>8.45</td>
<td>8.51</td>
<td>8.49</td>
<td>8.50</td>
<td>8.34</td>
</tr>
</tbody>
</table>
### Table 4.13: Grid search for $C_{10}S$

<table>
<thead>
<tr>
<th>C</th>
<th>$\sigma$</th>
<th>$2^{-10}$</th>
<th>$2^{-8}$</th>
<th>$2^{-6}$</th>
<th>$2^{-4}$</th>
<th>$2^{-2}$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-2}$</td>
<td>11.43</td>
<td>10.93</td>
<td>10.73</td>
<td>11.12</td>
<td>11.85</td>
<td>12.27</td>
<td></td>
</tr>
<tr>
<td>$2^2$</td>
<td>10.74</td>
<td>10.27</td>
<td>10.07</td>
<td>10.38</td>
<td>11.03</td>
<td>11.49</td>
<td></td>
</tr>
<tr>
<td>$2^4$</td>
<td>10.18</td>
<td>9.77</td>
<td>9.11</td>
<td>9.05</td>
<td>9.66</td>
<td>10.17</td>
<td></td>
</tr>
<tr>
<td>$2^6$</td>
<td>9.89</td>
<td>9.14</td>
<td>8.38</td>
<td>8.17</td>
<td>8.59</td>
<td>8.97</td>
<td></td>
</tr>
<tr>
<td>$2^8$</td>
<td>9.57</td>
<td>8.67</td>
<td>8.04</td>
<td>7.94</td>
<td>8.37</td>
<td>8.77</td>
<td></td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>9.20</td>
<td>8.46</td>
<td>7.92</td>
<td>7.92</td>
<td>8.32</td>
<td>8.82</td>
<td></td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>8.96</td>
<td>8.42</td>
<td>8.10</td>
<td>7.98</td>
<td>8.42</td>
<td>8.97</td>
<td></td>
</tr>
</tbody>
</table>

### Table 4.14: Grid search for $C_r S$

<table>
<thead>
<tr>
<th>C</th>
<th>$\sigma$</th>
<th>$2^{-10}$</th>
<th>$2^{-8}$</th>
<th>$2^{-6}$</th>
<th>$2^{-4}$</th>
<th>$2^{-2}$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-2}$</td>
<td>11.56</td>
<td>11.07</td>
<td>10.91</td>
<td>10.84</td>
<td>11.25</td>
<td>11.84</td>
<td></td>
</tr>
<tr>
<td>$2^2$</td>
<td>11.00</td>
<td>10.70</td>
<td>10.36</td>
<td>10.17</td>
<td>10.33</td>
<td>10.81</td>
<td></td>
</tr>
<tr>
<td>$2^6$</td>
<td>10.42</td>
<td>10.03</td>
<td>9.27</td>
<td>8.67</td>
<td>8.52</td>
<td>8.77</td>
<td></td>
</tr>
<tr>
<td>$2^8$</td>
<td>10.27</td>
<td>9.67</td>
<td>8.88</td>
<td>8.30</td>
<td>8.28</td>
<td>8.65</td>
<td></td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>10.12</td>
<td>9.28</td>
<td>8.64</td>
<td>8.27</td>
<td>8.29</td>
<td>8.82</td>
<td></td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>9.92</td>
<td>9.10</td>
<td>8.60</td>
<td>8.49</td>
<td>8.42</td>
<td>9.07</td>
<td></td>
</tr>
</tbody>
</table>

### Table 4.15: Grid search for $C_{10}M_{10}$

<table>
<thead>
<tr>
<th>C</th>
<th>$\sigma$</th>
<th>$2^{-10}$</th>
<th>$2^{-8}$</th>
<th>$2^{-6}$</th>
<th>$2^{-4}$</th>
<th>$2^{-2}$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-2}$</td>
<td>11.69</td>
<td>11.20</td>
<td>11.10</td>
<td>11.64</td>
<td>12.01</td>
<td>12.19</td>
<td></td>
</tr>
<tr>
<td>$2^2$</td>
<td>10.97</td>
<td>10.12</td>
<td>9.80</td>
<td>10.26</td>
<td>10.91</td>
<td>11.22</td>
<td></td>
</tr>
<tr>
<td>$2^4$</td>
<td>10.07</td>
<td>8.99</td>
<td>8.34</td>
<td>8.61</td>
<td>9.41</td>
<td>9.88</td>
<td></td>
</tr>
<tr>
<td>$2^6$</td>
<td>9.24</td>
<td>8.11</td>
<td>7.41</td>
<td>7.68</td>
<td>8.36</td>
<td>8.82</td>
<td></td>
</tr>
<tr>
<td>$2^8$</td>
<td>8.64</td>
<td>7.35</td>
<td>7.11</td>
<td>7.52</td>
<td>8.20</td>
<td>8.57</td>
<td></td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>7.90</td>
<td>7.24</td>
<td>7.15</td>
<td>7.55</td>
<td>8.17</td>
<td>8.52</td>
<td></td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>7.67</td>
<td>7.48</td>
<td>7.28</td>
<td>7.59</td>
<td>8.11</td>
<td>8.52</td>
<td></td>
</tr>
</tbody>
</table>

### Table 4.16: Grid search for $C_rM_{10}$

<table>
<thead>
<tr>
<th>C</th>
<th>$\sigma$</th>
<th>$2^{-10}$</th>
<th>$2^{-8}$</th>
<th>$2^{-6}$</th>
<th>$2^{-4}$</th>
<th>$2^{-2}$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-2}$</td>
<td>11.77</td>
<td>11.21</td>
<td>11.00</td>
<td>11.22</td>
<td>11.48</td>
<td>11.74</td>
<td></td>
</tr>
<tr>
<td>$2^2$</td>
<td>11.04</td>
<td>10.15</td>
<td>9.63</td>
<td>9.70</td>
<td>9.86</td>
<td>10.29</td>
<td></td>
</tr>
<tr>
<td>$2^4$</td>
<td>10.17</td>
<td>9.23</td>
<td>8.76</td>
<td>8.56</td>
<td>8.56</td>
<td>8.90</td>
<td></td>
</tr>
<tr>
<td>$2^6$</td>
<td>9.45</td>
<td>8.85</td>
<td>8.44</td>
<td>8.07</td>
<td>7.91</td>
<td>8.14</td>
<td></td>
</tr>
<tr>
<td>$2^8$</td>
<td>9.05</td>
<td>8.53</td>
<td>8.03</td>
<td>7.70</td>
<td>7.8</td>
<td>8.01</td>
<td></td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>8.90</td>
<td>8.25</td>
<td>7.80</td>
<td>7.79</td>
<td>7.79</td>
<td>8.03</td>
<td></td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>8.62</td>
<td>8.18</td>
<td>7.93</td>
<td>7.92</td>
<td>7.86</td>
<td>8.07</td>
<td></td>
</tr>
</tbody>
</table>
### 4.4. Setting the parameters for SVM

#### Table 4.17: Grid search for $SM_{10}$

<table>
<thead>
<tr>
<th>$C$</th>
<th>$\sigma$</th>
<th>$2^{-10}$</th>
<th>$2^{-8}$</th>
<th>$2^{-6}$</th>
<th>$2^{-4}$</th>
<th>$2^{-2}$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-2}$</td>
<td></td>
<td>11.48</td>
<td>10.93</td>
<td>10.77</td>
<td>11.33</td>
<td>11.71</td>
<td>12.05</td>
</tr>
<tr>
<td>$2^{-1}$</td>
<td></td>
<td>10.67</td>
<td>9.78</td>
<td>9.40</td>
<td>9.63</td>
<td>10.14</td>
<td>10.73</td>
</tr>
<tr>
<td>$2^0$</td>
<td></td>
<td>9.79</td>
<td>8.95</td>
<td>8.33</td>
<td>8.10</td>
<td>8.42</td>
<td>8.90</td>
</tr>
<tr>
<td>$2^1$</td>
<td></td>
<td>9.10</td>
<td>8.40</td>
<td>7.68</td>
<td>7.33</td>
<td>7.46</td>
<td>7.70</td>
</tr>
<tr>
<td>$2^2$</td>
<td></td>
<td>8.76</td>
<td>7.89</td>
<td>7.29</td>
<td>7.34</td>
<td>7.43</td>
<td>7.60</td>
</tr>
<tr>
<td>$2^3$</td>
<td></td>
<td>8.32</td>
<td>7.57</td>
<td>7.37</td>
<td>7.35</td>
<td>7.57</td>
<td>7.63</td>
</tr>
<tr>
<td>$2^4$</td>
<td></td>
<td>7.97</td>
<td>7.65</td>
<td>7.57</td>
<td>7.49</td>
<td>7.71</td>
<td>7.75</td>
</tr>
</tbody>
</table>

#### Table 4.18: Grid search for $C_{10}SM_{10}$

<table>
<thead>
<tr>
<th>$C$</th>
<th>$\sigma$</th>
<th>$2^{-10}$</th>
<th>$2^{-8}$</th>
<th>$2^{-6}$</th>
<th>$2^{-4}$</th>
<th>$2^{-2}$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-2}$</td>
<td></td>
<td>11.49</td>
<td>11.02</td>
<td>11.15</td>
<td>11.93</td>
<td>12.29</td>
<td>12.53</td>
</tr>
<tr>
<td>$2^{-1}$</td>
<td></td>
<td>10.60</td>
<td>9.69</td>
<td>9.76</td>
<td>10.62</td>
<td>11.26</td>
<td>11.68</td>
</tr>
<tr>
<td>$2^0$</td>
<td></td>
<td>9.57</td>
<td>8.58</td>
<td>8.19</td>
<td>8.91</td>
<td>9.73</td>
<td>10.24</td>
</tr>
<tr>
<td>$2^1$</td>
<td></td>
<td>8.74</td>
<td>7.65</td>
<td>7.20</td>
<td>7.81</td>
<td>8.59</td>
<td>8.99</td>
</tr>
<tr>
<td>$2^2$</td>
<td></td>
<td>8.19</td>
<td>7.14</td>
<td>6.93</td>
<td>7.58</td>
<td>8.33</td>
<td>8.73</td>
</tr>
<tr>
<td>$2^3$</td>
<td></td>
<td>7.51</td>
<td>7.06</td>
<td>6.93</td>
<td>7.63</td>
<td>8.34</td>
<td>8.80</td>
</tr>
<tr>
<td>$2^4$</td>
<td></td>
<td>7.47</td>
<td>7.36</td>
<td>6.99</td>
<td>7.63</td>
<td>8.42</td>
<td>8.89</td>
</tr>
</tbody>
</table>

#### Table 4.19: Grid search for $C_SM_{10}$

<table>
<thead>
<tr>
<th>$C$</th>
<th>$\sigma$</th>
<th>$2^{-10}$</th>
<th>$2^{-8}$</th>
<th>$2^{-6}$</th>
<th>$2^{-4}$</th>
<th>$2^{-2}$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-2}$</td>
<td></td>
<td>11.40</td>
<td>10.93</td>
<td>10.95</td>
<td>11.52</td>
<td>11.88</td>
<td>12.19</td>
</tr>
<tr>
<td>$2^{-1}$</td>
<td></td>
<td>10.63</td>
<td>9.84</td>
<td>9.53</td>
<td>10.00</td>
<td>10.53</td>
<td>11.07</td>
</tr>
<tr>
<td>$2^0$</td>
<td></td>
<td>9.80</td>
<td>8.93</td>
<td>8.39</td>
<td>8.31</td>
<td>8.89</td>
<td>9.46</td>
</tr>
<tr>
<td>$2^1$</td>
<td></td>
<td>9.08</td>
<td>8.38</td>
<td>7.47</td>
<td>7.35</td>
<td>7.76</td>
<td>8.17</td>
</tr>
<tr>
<td>$2^2$</td>
<td></td>
<td>8.64</td>
<td>7.63</td>
<td>7.12</td>
<td>7.22</td>
<td>7.51</td>
<td>7.91</td>
</tr>
<tr>
<td>$2^3$</td>
<td></td>
<td>8.23</td>
<td>7.29</td>
<td>7.17</td>
<td>7.19</td>
<td>7.59</td>
<td>7.95</td>
</tr>
<tr>
<td>$2^4$</td>
<td></td>
<td>7.70</td>
<td>7.52</td>
<td>7.28</td>
<td>7.45</td>
<td>7.65</td>
<td>8.01</td>
</tr>
</tbody>
</table>
4.5 Lifetime prediction experiments

Based on the results presented in section 4.3 and 4.4, models were trained with the selected design parameters. This section contains the prediction performance results of these final models on the previously unused test sets, comprising 20% of the original datasets.

Figure 4.2 displays the lifetime prediction performance of the models trained and tested on data from all five repositories, i.e. one MLP and one SVM model for each feature group combination. For comparison, the error of a baseline prediction (BP) is displayed as a black line in the diagram. The baseline prediction consists of outputting the mean lifetime of the training set for each input instead of making a prediction with a machine learning model. The purpose of the baseline prediction is to act as the simplest possible prediction and can be used to see to what extent the MLP and SVM models actually improve the performance. The right Y-axis in each diagram expresses the prediction RMSE as a percentage of the baseline prediction error. Similarly, figures 4.3, 4.4, 4.5, 4.6 and 4.7 display the performance of models trained and tested on single repositories.

Considering the experiments performed on data from all five repositories in figure 4.2, the best performance was achieved when all three feature groups were included. This meant an RMSE of 13.84 commits for MLP and 12.90 commits for SVM. Compared to the RMSE of the baseline prediction at 20.44 commits, this corresponds to 67.7% for MLP and 63.11% for SVM, i.e. significantly better than outputting the mean lifetime of the training set.

![Figure 4.2: Performance on data from all repositories. The X-axis shows different combinations of the feature group variants. The baseline prediction (BP) error, obtained by outputting the mean lifetime of the training set, is displayed as the black line. The right Y-axis expresses the prediction RMSE as a percentage of the baseline prediction error.](image)

In figures 4.2-4.7 it can be seen that all datasets display a common pattern in how the performance of the feature groups relate to each other. In general, the best performance is obtained from the $C_{10}SM_{10}$ combination. The exception is the Dropwizard repository (figure

---

In figures 4.2-4.7, it can be seen that all datasets display a common pattern in how the performance of the feature groups relate to each other. In general, the best performance is obtained from the $C_{10}SM_{10}$ combination. The exception is the Dropwizard repository (figure
4.5. Lifetime prediction experiments

which displays the best performance with $C_{10}M_{10}$. For all datasets, $M_{10}$ is the feature group that performs best alone.

The error magnitude of the best performance differs between different datasets. The Dropwizard repository yields the smallest errors of 4.42 commits for MLP and 4.16 commits for SVM. That is considerably lower than the Hystrix repository, which is second best with an MLP error of 7.18 commits and an SVM error of 6.32 commits. The Gitblit repository, on the other hand, has the largest errors of 18.54 commits for MLP and 17.90 commits for SVM.

Figure 4.3: Performance on data from the Retrofit repository.
4.5. Lifetime prediction experiments

Figure 4.4: Performance on data from the Hystrix repository.

Figure 4.5: Performance on data from the Gitblit repository.
4.5. Lifetime prediction experiments

Figure 4.6: Performance on data from the Dropwizard repository.

Figure 4.7: Performance on data from the Okhttp repository.
Figure 4.8 shows the best MLP and SVM performance on each of the datasets. When comparing the best performance on each dataset, SVM gave a smaller error than MLP in all cases. The prediction error is roughly between 4 and 19 commits for both MLP and SVM, depending on dataset.

![Best performances](image)

Figure 4.8: The best performance on each dataset.
In this chapter, certain parts of the study are discussed more closely. Section 5.1 looks into a few aspects of the method and section 5.2 deals with the results obtained from the experiments in the previous chapter. Finally, section 5.3 reflects on the possible effects of this study.

5.1 Method

In this section, a few aspects of the method are discussed. This includes the definitions of lifetime and pieces of code (section 5.1.1), the features used for describing a piece of code (section 5.1.2) and the way in which the experiments were conducted (section 5.1.3).

5.1.1 Definitions of lifetime and pieces of code

The intuitive idea of predicting code lifetime proved to be rather difficult, mostly due to the definitions of lifetime and code pieces that were made in order to define the task. The notion of lifetime used in this study was not well-suited for Git repositories with multiple branches. Branches caused the calculation of lifetime values to be approximate, as code merged in from other branches appeared to be added in the merge commit. Therefore, the values regarded as correct in the learning procedures were not entirely correct in all cases, which lowers the validity of the predictions. It was not investigated to what extent this had an impact on the lifetime calculations.

Another consequence of the lifetime definition was that only code that had actually been modified was considered, resulting in truncation of the data. As shown in the extraction of code pieces in table 4.1, there is a fair amount of code that remains untouched after being added to the repository. As a consequence, the predictions do not apply to all code that is written in a project. It may also have been possible to make use of the unmodified pieces with a different study design, since they have a known lower bound for their lifetime and features that may reflect this property.

Most of the features in the change metadata group were not mappable to a specific Git hunk, but rather to a Git commit, which may not have been optimal in describing individual pieces of code as defined in this study. This suggests that entire Git commits, instead of parts of it, could be selected as pieces of code, as previously done by Mockus and Weiss [1] as well as Kim, Whitehead Jr., and Zhang [2] in fault prediction tasks. The approach taken in
5.2. Results

This study was based on a desire to make as specific predictions as possible, but it was not investigated which effect this had on the results compared to other code piece formats as no other format was tested.

5.1.2 Features

Some of the change metadata features were calculated with respect to the single file in which a particular piece of code resided. For example, the developer experience was the number of previous commits including that specific file, and developer count was the number of developer involved in modifying that file. This can be compared to calculating them with respect to the entire repository, which was done by Mockus and Weiss [1]. Even though they predicted faults in entire changesets (commits), this approach could perhaps still be useful for smaller pieces of code.

It may also be possible that some of the features included in the input vectors made no contribution of unique information to the learning algorithms, but instead decreased the effectiveness of learning. These features could have been filtered out, e.g. by principal component analysis, as suggested by LeCun, Buttolu, Orr, and Müller [16].

5.1.3 Experiments

The MLP and SVM design parameters were selected by measuring the performance on data from only one repository (Hystrix), which may have lead to parameters that were biased towards that particular data. The reason for this approximation approach was purely a time issue, since training and validation was a time-consuming process. For possibly improved results, all the datasets could have been used in separate cross-validation processes for selecting parameters.

The dataset composed of data from all five repositories was created by merging the other datasets and shuffling the instances. From this compound dataset, 20% of the data was used for testing and the remaining 80% for training and validation. A perhaps more ideal setup could have been to train and validate on four repositories and use the fifth for testing. In this manner, the entire test repository would remain unseen until the testing phase, without giving the machine learning algorithms the advantage of being exposed to e.g. repository-specific code terms during training. Whether or not the taken approach was a weakness depends on how lifetime prediction should be applied; for making predictions for the same repositories as those used as training data or for other repositories.

5.2 Results

This section concerns the results presented in chapter 4 – Results. Section 5.2.1 focuses on the lifetime distributions of the different datasets and section 5.2.2 discusses the results of the lifetime prediction experiments.

5.2.1 Lifetime distributions of datasets

As shown in table 4.2 in section 4.1 mean lifetime varies largely between the different datasets, which suggests that different source code projects have different behavior related to code changes and version control. This is exemplified by Gitblit and Dropwizard, which stand out as the repositories with the longest and shortest mean lifetime of 16.37 and 4.78 commits, respectively. Judging from the large differences in mean lifetime and standard deviation between the repositories, these characteristics are not universal for software projects.

The empirical CDF plot (figure 4.1 in section 4.1) further shows the different characteristics of the repositories. The two extreme cases are Dropwizard, in which all code pieces have a
5.2. Results

lifetime of at most 40 commits, and Gitblit, in which a few percent of the code pieces have a lifetime between 100 and 200 commits.

5.2.2 Lifetime prediction experiments

Based on the results obtained with data from all repositories (figure 4.2 in section 4.5), all three feature groups are useful in lifetime prediction. This is because at least one variant of each feature group alone gives a smaller error than the baseline prediction. The best predictions using a single feature group were made with the $M_{10}$ variant. The importance of including log message terms in the change metadata features can be seen by comparing the results obtained from different filtering ($M_0$, $M_1$, $M_{10}$). The error decreases when going from 0 to 1 % and decreases further when 10 % are used. However, the largest improvement in performance is made when going from 0 to 1 %.

As for code contents, $C_{10}$ gave significantly better results than $C_1$ or $C_r$, which shows that a higher number of terms improves the performance at the cost of more complex models. Predictions based on $C_r$ alone yielded a larger error than the baseline prediction, which may suggest that the reserved code terms are useless in lifetime prediction. On the other hand, both $C_{10}SM_{10}$ and $C_rSM_{10}$ gave slightly improved results over $SM_{10}$ alone, which shows that both $C_{10}$ and $C_r$ are useful.

From the experiments on all datasets (figures 4.2-4.7 in section 4.5), it can be noted that in general, some combinations of two feature groups perform almost as well as combinations of three. The improved performance of using a third feature group comes at the cost of a significantly higher model complexity and more time needed for extracting features. If applying lifetime prediction in practice, using only two feature groups may be a simpler solution with a performance that is close to the best possible.

As seen in figures 4.2-4.7 in section 4.5, MLP generally yields a smaller error than SVM when using individual feature groups. However, the smallest error on each dataset, which is the main interest of this study, was obtained with SVM and (mostly) three feature groups. Based on this, SVM can be considered the most suitable of the two algorithms for lifetime prediction.

As can be seen by comparing the results for the different datasets, the magnitude of the prediction error differs quite a lot between different repositories (for example Dropwizard and Gitblit). A possible explanation of these large differences can be found in the empirical CDF plot in figure 4.1 in section 4.1. It can be seen that the Dropwizard repository has the smallest variation in lifetime values, while the values in the Gitblit repository are distributed over a much larger range. The curves for the other repositories reside somewhere in between the two. This corresponds well to the relation between the magnitudes of the prediction errors for the different repositories, which suggests that lifetime prediction performance improves as the variation in lifetime values decreases. The performance of lifetime prediction is therefore highly dependent on the dataset.

Despite the small RMSE obtained from predictions on the Dropwizard repository, the proportional improvement over the baseline prediction was not larger than for other repositories. With an RMSE for the MLP at 87.0 % of the baseline prediction error, this is the smallest proportional improvement for MLP predictions on any dataset. It also has smallest the improvement for SVM at 82.3 % of the baseline prediction error. The best MLP predictions in terms of proportional improvement over the baseline error (59.4 % of baseline) were made on the Gitblit repository, using all three feature groups. For SVM, the best proportional improvement (53.4 % of baseline) was obtained on the Hystrix repository, also using all three feature groups.

Judging from the error magnitude of the best performances (figure 4.8 in section 4.5), lifetime prediction performs worse on the dataset composed all five repositories than on the individual repositories (with the exception of Gitblit). However, considering error relative to the baseline prediction, the experiment using data from all repositories yields better results.
than any of Retrofit, Dropwizard or Okhttp separately. This suggests that the repositories have common feature patterns that correlate with lifetime, despite their differences.

Due to the nature of the output value (lifetime), the results are difficult to compare to the related studies presented in section 1.4, as those focus on predicting faults in code. It can, however, be concluded that the same kinds of features are useful for predicting both faults and lifetime, as the features used for predicting lifetime were based on those used for predicting faults. Note that this does not mean that lifetime for a piece of code correlates with its fault-proneness.

5.3 The work in a wider context

If functioning ideally, lifetime prediction could be used as a complement to existing tools for code evaluation. This could potentially have an influence on anyone working with software development and particularly the software industry. Possible effects of using lifetime predictions includes more effective elimination of weak code and greater knowledge of code life cycles and modifications. The former presumes that code lifetime is related to traditional measures of code quality, which was a hypothesis on which this work was based. Whether or not there actually exists one such connection was not investigated in this study.
6 Conclusion

This chapter concludes the findings of this study. Section 6.1 responds to the research questions that were stated in section 1.1. Section 6.2 makes a few suggestions for future work on the subject.

6.1 Conclusions

How precisely can the lifetime of a piece of code be predicted? In the experiments it was shown that the RMSE was 13.84 Git commits for MLP and 12.90 commits for SVM when using data from all five repositories. This was significantly better than the baseline prediction error of 20.44 commits and supports the idea that lifetime to some extent can be predicted based on information that can be extracted from the code. On the other hand, the mean lifetime of the dataset was 11.16 commits, so the predictions are likely too inaccurate to be used for code evaluation in practice.

The best results were obtained when using training and testing data from single repositories. The repository with the lowest RMSE was Dropwizard with 4.42 commits for MLP and 4.16 for SVM. However, if a single-repository approach would be applied in practice, a repository needs to contain a sufficient amount of training data before any predictions can be made.

From the experiments it also became evident that performance varies largely depending on the characteristics of the dataset, which further devalues the applicability of lifetime prediction. The findings of this study may therefore not have any direct effect on developers using evaluation tools, which was the intended target group for lifetime predictions.

All three feature groups used in this study (code contents, static code analysis and change metadata) proved to be useful for lifetime prediction. The code contents were most successfully represented by the 10 % most frequent terms, which yielded better results than the 1 % most frequent terms or a set of reserved terms in the Java language. As for change metadata, the best performance was achieved when using the 10 % most frequent log message terms as features, as opposed to 0 or 1 %.

In all the experiments on the different datasets (i.e. all data and each repository separately), SVM performed better than MLP when all features were included. Given the feature groups and data used in this study, SVM can be considered the most suitable of the two for lifetime prediction.
6.2 Future work

Regarding the aim of this study, two particular aspects could be explored further in future work. Firstly, the effects of alternative definitions of lifetime and pieces of code on prediction performance could be investigated. For example, lifetime could be measured in actual time and code pieces could represent entire Git commits.

Secondly, other features could be included in addition to those used in this study. Examples of additional features include change metadata metrics calculated on a repository level instead of the current single-file approach.

In a wider perspective, predicting code lifetime may not be limited to predicting numerical values in order to be useful. From a developer’s perspective, it could even be more useful to know that piece X is likely to have a shorter lifetime than piece Y, instead of a numerical lifetime for X that is off by 20 commits. Thus, an interesting subject for future studies could be to investigate how accurately code pieces can be ranked relative to each other with respect to lifetime.
Bibliography


[33] retrofit. URL: [https://github.com/square/retrofit](https://github.com/square/retrofit) (visited on 08/03/2016).

[34] Hystrix. URL: [https://github.com/Netflix/Hystrix](https://github.com/Netflix/Hystrix) (visited on 08/03/2016).


[37] okhttp. URL: [https://github.com/square/okhttp](https://github.com/square/okhttp) (visited on 08/03/2016).


[40] *PMD*. URL: http://pmd.github.io/pmd-5.4.2/index.html (visited on 08/03/2016).
Most frequent features

In the following sections, the ten most frequently occurring features for code pieces in the different repositories are shown. Code terms are listed in section A.1, PMD warnings in section A.2, and log message terms in section A.3.

A.1 Code terms

Table A.1 shows the ten most frequent code terms for code pieces in each repository and the number of occurrences.

<table>
<thead>
<tr>
<th>Repository</th>
<th>Terms and number of occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrofit</td>
<td>) (21296), ( (20586), . (18720), ; (11603), ” (10286), { (6230), , (6153), } (4303), @ (4120), = (3921)</td>
</tr>
<tr>
<td>Hystrix</td>
<td>. (44902), ) (42817), ( (41662), ; (22784), , (17725), command (14563), hystrix (13459), get (11828), ” (9305), { (9057),</td>
</tr>
<tr>
<td>Gitblit</td>
<td>. (77970), ) (67413), ( (64378), ; (42337), , (31111), ” (30645), , (16967), = (16247), get (15054), * (14339),</td>
</tr>
<tr>
<td>Dropwizard</td>
<td>. (43118), ) (29640), ( (28627), ; (18285), ” (9595), , (6729), } (5745), import (5412), get (4822), &lt; (4697),</td>
</tr>
<tr>
<td>Okhttp</td>
<td>) (81503), ) (79039), ( (77504), ; (47093), ” (39053), , (31196), = (15242), ) (14965), url (11362), connection (11043)</td>
</tr>
<tr>
<td>All repositories</td>
<td>. (263749), ) (242669), ( (232757), ; (142102), ” (98884), , (92914), } (52964), = (45531), get (44447), ) (39697)</td>
</tr>
</tbody>
</table>

Table A.1: The ten most frequent code terms in each repository.
### A.2 PMD warnings

Table A.2 shows the ten most frequent PMD warnings for code pieces in each repository and the number of occurrences.

<table>
<thead>
<tr>
<th>Repository</th>
<th>Warnings and number of occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrofit</td>
<td>CommentRequired (10346), TooManyMethods (3931), GodClass (3200), MethodArgumentCouldBeFinal (2737), DataflowAnomalyAnalysis (2716), AtLeastOneConstructor (2526), LocalVariableCouldBeFinal (2510), ExcessivePublicCount (1778), TooManyStaticImports (1775), CommentDefaultAccessModifier (1610),</td>
</tr>
<tr>
<td>Hystrix</td>
<td>CommentRequired (10408), TooManyMethods (7609), DataflowAnomalyAnalysis (6847), JUnitAssertionsShouldIncludeMessage (5932), GodClass (5778), ExcessiveClassLength (4846), ExcessivePublicCount (4620), MethodArgumentCouldBeFinal (3543), LocalVariableCouldBeFinal (3157), TooManyStaticImports (2425),</td>
</tr>
<tr>
<td>Gitblit</td>
<td>DataflowAnomalyAnalysis (46376), CommentRequired (23366), GodClass (11012), LocalVariableCouldBeFinal (9832), TooManyMethods (8940), MethodArgumentCouldBeFinal (5425), ExcessivePublicCount (4236), ExcessiveClassLength (4024), TooManyFields (3118), UseUtilityClass (3066),</td>
</tr>
<tr>
<td>Dropwizard</td>
<td>CommentRequired (12682), AtLeastOneConstructor (3999), MethodArgumentCouldBeFinal (2028), DataflowAnomalyAnalysis (1750), TooManyMethods (1512), CommentSize (808), GodClass (655), LocalVariableCouldBeFinal (647), TestClassWithoutTestCases (609), TooManyStaticImports (543),</td>
</tr>
<tr>
<td>Okhttp</td>
<td>CommentRequired (25082), TooManyMethods (19120), GodClass (15961), TooManyStaticImports (9764), LocalVariableCouldBeFinal (8325), ExcessivePublicCount (7943), DataflowAnomalyAnalysis (7625), AtLeastOneConstructor (7606), ExcessiveClassLength (6030), JUnitAssertionsShouldIncludeMessage (5944),</td>
</tr>
<tr>
<td>All repositories</td>
<td>CommentRequired (81884), DataflowAnomalyAnalysis (63314), TooManyMethods (41112), GodClass (36606), LocalVariableCouldBeFinal (24471), MethodArgumentCouldBeFinal (19622), ExcessivePublicCount (18908), AtLeastOneConstructor (17821), ExcessiveClassLength (15733), TooManyStaticImports (14620),</td>
</tr>
</tbody>
</table>

Table A.2: The ten most frequent PMD warnings in each repository.
# A.3 Log message terms

Table A.3 shows the ten most frequent log message terms for code pieces in each repository and the number of occurrences.

<table>
<thead>
<tr>
<th>Repository</th>
<th>Terms and number of occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrofit</td>
<td>to (595), for (401), converter (240), add (225), and (212), the (212), request (175), factory (170), call (159), a (157)</td>
</tr>
<tr>
<td>Hystrix</td>
<td>hystrix (655), to (632), and (407), added (389), command (378), for (324), metrics (267), thread (222), of (208), the (205)</td>
</tr>
<tr>
<td>Gitblit</td>
<td>and (1473), to (1471), for (840), repository (555), in (538), the (503), of (495), git (472), page (459), user (438)</td>
</tr>
<tr>
<td>Dropwizard</td>
<td>to (1441), the (814), move (759), for (565), from (478), com (467), codahale (466), in (410), tests (393), add (389)</td>
</tr>
<tr>
<td>Okhttp</td>
<td>to (1475), and (840), http (774), the (719), in (550), for (540), ok (534), response (472), a (451), use (413)</td>
</tr>
<tr>
<td>All repositories</td>
<td>to (5614), and (3274), for (2670), the (2453), in (1782), a (1491), of (1439), from (1327), move (1136), add (1076)</td>
</tr>
</tbody>
</table>

Table A.3: The ten most frequent log message terms in each repository.