Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BM-2008-12

Master

im Studiengang "Angewandte Informatik"

Detection of feature freezes using clustering algorithms

Steffen Herbold

am Lehrstuhl für

Softwaretechnik und Verteilte Systeme

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

26. September, 2008

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

Tel. +49 (5 51) 39-1 44 14
Fax +49 (5 51) 39-1 44 15
Email office@informatik.uni-goettingen.de
WWW www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 26. September, 2008

1

Georg-August-Universität Göttingen

Masterthesis

Detection of feature freezes using clustering algorithms

Steffen Herbold

26.09.2008

Betreut durch Prof. Dr. Jens Grabowski
Institut für Informatik
Georg-August-Universität Göttingen

*Thanks to everyone who provided help and suggestions for this thesis, especially to Prof. Dr. Waack, Prof. Dr. Grabowski and Dr. Neukirchen.

Contents

1. Introduction

Software development is a complex process which cannot be easily quantified. Thus, the analysis of a development process or even a single project is very difficult. Usually experts take a look at what happened retrospectively, maybe perform a review and try to improve the process by implementing countermeasures to the mistakes that were made. Managers usually consider simple economical metrics such as the amount of time and money spent on the project or the number of developers involved. Of course they also consider whether the project was successful or not. However, these attributes only consider management aspects of the project, but not the software itself. Attributes like the number of bugs or the size of the project are also important. The problem is that software is abstract. How is the complexity of software defined? How is its size defined? What is good software anyway?

One approach to answer this question are software metrics. They try to quantify attributes of the software itself, like the complexity, the reusability or the size. But as software is something abstract, it is not easy to define a software metric that is accepted. Every software metric has advantages and disadvantages, no metric is foolproof. However, during the last years software metrics were more and more accepted and included in popular development tools to assist software development. Microsoft included support for software metrics in Microsoft Visual Studio 2008. To measure Java code directly using Eclipse, for example the Eclipse Metrics Plug-in [3] exists. The next logical step is to use the software metrics to make management decisions. This has always been done implicitly: whether a project is ready for release or not depends on the number of bugs and the number of completed features. Both are software metrics.

Independent of this, the field of machine learning has infiltrated all kinds of sciences and industrial work: genealogists use it to analyse traits of the human genome, financial institutes use it to predict if a client is credit worthy, computer scientist use machine learning to improve spam filters. So why not use machine learning techniques to analyse the quality of software or even the process of software development? This is no new idea. PROMISE [5] for example gathers metric data about software so that it can be used for analysis. In a yearly workshop different approaches to analyse this data are presented. Most of these approaches try to detect modules that need extensive testing or they try to determine the probability that a module has a bug.

The goal of this work is to show that software metrics and machine learning techniques can also be used to analyse the process of software development itself. To perform such an analysis, sets of metric data from different points of time during the project are needed. One major part of this thesis addresses a way to mine software repositories to gather metric data of already

finished projects. While this is an easy task for small projects, the amount of work increases greatly for big projects. Furthermore, software tools are needed for various task: measuring the software metrics; the processing of the gathered metric data into a format that can be used for analysis; the analysis itself.

As an example for a way to mine metric data as well as to show that software metrics and machine learning techniques can indeed be used to analyse the process of software development, data from the development process of the Eclipse Platform Project 3.2 and the Eclipse Java Development Tools 3.2 are mined. This data is then used to detect a *feature freeze* that occurred during both projects. To perform the analysis, the k-means clustering algorithm is used. This is a rather simple approach. However, the collection and preparation of the metric data itself is not simple. Preparation is needed because several problems would occur if the unprocessed metric data were be used as input for the k-means algorithm. One of these problems is that it does not make sense to use an arbitrary set of metrics, the metrics have to be chosen carefully instead.

After the introduction, the second chapter gives the foundations on which this work is built. It introduces basic concepts of software engineering and the machine learning techniques used in this thesis. Chapter 3 shows how metric data can be mined from code versioning systems such as SVN and CVS and bugtracking systems like Bugzilla. Because the result of the mining can not be used for analysis without great effort, in chapter 4 a tool to preprocess the mined data is introduced. In chapter 5 a practical experiment to mine metric data in the way it has been shown in chapter 3 is performed and a way to analyse the data using the k-means clustering algorithm is shown. Chapter 6 concludes the work and gives an outlook on possibilities for future research.

2. Basic definitions and concepts

Because this thesis combines software engineering with the machine learning field from the theoretical computer science, a common basis has to be established first. This way people from both of these fields have a better understanding of the later parts of the thesis.

As a short introduction to software engineering, definitions for the terms and concepts that are used later on in this thesis are provided. A more detailed introduction can be found in the literature [15]. Afterwards some basic concepts of machine learning are established and the algorithms that are used as part of this work are introduced.

2.1. Software Engineering and Software Metrics

The first question one has to answer is: what is software engineering? According to the IEEE it is “the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software” [6]. As this general defintion shows, software engineering is a wide field. The focus of this work is on quality assessment of specific features of a software development process. To analyse the process, data about projects are used retrospectively. This is feasible, because a process is a general description how a project should be planed and executed. What kind of data are used will be discussed later on.

First of all, a common definition of project is needed. In [11] the following definition is used:

Definition 2.1. Software Project
A project is a complex scheme which is

- chronologically bounded through a defined start and end date
- defined through the uniqueness of its terms and conditions like the project goals, the project outline, the people and resources et cetera.

For the analysis done as part of this thesis, the defined start and end are important. If one wants to analyse the progress of a project, knowledge about the start and end is required. Consider a project where the beginning is not known. In case the project was a totally new development it does not matter when the progress of the source code shall be analysed, because at the project start there was no source code. However, many projects reuse old code and build new versions of an already existing piece of software. In this case, one has to have

knowledge about the code basis. Otherwise the old code that was developed as part of another project would also be considered as part of this project. This would falsify the results. If the time that the development took is part of the analysis it is also mandatory to know when the project started. Similar arguments can be said about the end of the project. If the final status is not known, it is difficult to say something about the progress. This is well known, hence some people say that the last 10 % of a project require 90 % of the resources. This may be exaggerated, but it has a true core.

As a project progresses there are often versions of a special importance. These versions are commonly called *milestones*. The following definition of milestones is translated from [11].

Definition 2.2. Milestone
A milestone is the achievement of a measurable, meaningful event during a project at a fixed and planned point of time.

The problem with the term milestone is that it is used for two purposes. The first is as it is in the definition – a general name for important events during a project. On the other hand, these points themselves are sometimes called milestones. So when using this term is used, one has to think about the context it is used in. In most cases when a milestone takes place, a version of the software is build and named after the milestone. This is another problem. The commonly known Alpha and Beta Versions usually mark a milestone of the project and the version is the result of that milestone. Other common milestones are so called *Release Candidates*. A Release Candidate is usually build near the end of a project, when the development is already frozen and only testing and bugfixing is done. Such a Release Candidate is usually also available to a bigger audience than just the development team, to find more failures in the software.

A specific feature of a project that is usually a milestone is a *feature freeze*. A feature freeze marks the end of the development of new features. Afterwards the focus of the project changes to stabilizing the project, such as testing and documentation. Obviously every project has to have a feature freeze, even if the feature freeze is the same as the end of the project. Usually this is not the case, but it depends on the kind of the development process that is used. If one considers *extreme programming* [7] or other agile process models in which there are short development cycles, those modells do not need such a thing as a feature freeze. However, most non-agile process models implement something like a feature freeze.

Up till now, only software engineering and projects in general were discussed and it was hinted at the usage of “data” about these projects. However, nothing was said about the kind of data. One way to obtain data about software in general is to use *metrics*. A metric uses some kind of measure to describe distance. In mathematics, the following definition is used for a measure.

Definition 2.3. A measure M is a well-defined map

$$\begin{aligned} M :& A_M \to V \\ & M(x) \mapsto v \end{aligned} \tag{2.1}$$

where A_M is the attribute space and V is the space of measurement values. The value v is the measure of the entity x.

The general meaning of this formal definition is quite simple: Each attribute is mapped to a value. While the definition uses abstract spaces for many metrics the space of measurement values are the positive reals or integers. But other spaces are also possible, as we will discuss later. The definition that Fenton and Pfleeger use in [8] is not so formal.

Definition 2.4. Measure (Fenton/Pfleeger)
Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them accordingly to clearly defined rules.

A measure is not yet a metric. A measure is only a way to assign a value to an attribute. However, metrics use measures. Again, first the mathematical definition of a metric is considered.

Definition 2.5. Metric (Mathematical)
A metric is a map

$$\begin{aligned} m :& E \times E \to \mathbb{R} \\ & (x, y) \mapsto m(x, y), \end{aligned} \tag{2.2}$$

such that

(i) $m(x, x) = 0 \; \forall \; x \in E$

(ii) $m(x, y) \geq 0 \; \forall \; x, y \in E$

(iii) $m(x, y) = m(y, x) \; \forall \; x, y \in E$

(iv) $m(x, z) \leq m(x, y) + m(y, z) \; \forall \; x, y, z \in E,$

that defines the distance between elements of a space E.

For software metrics this formal definition is too tight. For example, if a measure did just define a map to values such as “Good” and “Bad” it would not be possible to define a distance, as the mathematical definition requires. Thus, for software metrics a different definition is required. The IEEE defines the term in the standard 610.12 [12].

Definition 2.6. Software metric (IEEE 610.12)
A metric is a quantitative measure of the degree to which a system, component, or product possesses a given attribute. A quality metric is

1. a quantitative measure of the degree to which an item possesses a given quality attribute.
2. a function whose input and whose output is a single numerical value that can be interpreted as the degree to which the software possesses a given quality attribute.

This definition is more vague and the similarity to definition 2.4 of a measure is greater. With the following definition of relation systems and representatives another way to interpret metric results is introduced.

Definition 2.7. Relation Systems
A empirical relation system is a system in which attributes are not measured, but only compared.

A numerical relation system is a numerical space (i.e. $\mathbb{R}_+$ or the $\mathbb{Z}_+$ to which attributes can be mapped using a measure M).

A numerical relation system is called representative for a empirical relation R, if the according relation R' is fulfilled in the numerical system.

To understand what this definition means, consider the human attribute of height. In the empirical system, one human may be larger than another. A numerical system for height is a representative for this empirical system if the larger human has a higher numerical value, for example the larger human has a height of 180 cm and the smaller of 175 cm. The term of representative systems is general for metrics. If you consider it for software metrics, it means that if you know for some reason that one project is larger than another, a metric that measures the size has to have the same result. Consequently, if a quality metric yields a better result for one project than another that project should really have a higher quality. This is a difficult subject because terms like quality often depend on the point of view of the observer. For a manager a project has high quality if it has no failures, while a developer also considers the structure of the source code. Software metrics discard the point of view and are neutral.

There are two general types of software metrics, the direct and the indirect metrics.

Definition 2.8. Direct and indirect metric
A direct software metric is a metric defined over a measure that can be directly calculated. An indirect metric is a metric defined over a measure that is indirectly calculated using other metrics.

But why is it important to have both direct and indirect metrics? The need for direct metrics is obvious. Without direct metrics, there would be no metrics at all. But why indirect metrics? One may argue that these metrics are not plausible, when they are not directly measurable. However, considering that an indirect metric is calculated using other (direct or indirect) metrics, it is possible to calculate all metrics directly. This direct computation might be very complex, but in the end all metrics are based on direct metrics. But if they can be measured directly, why use indirect metrics at all? The two most important reasons are simplicity and interrelationship.

For example, consider an indirect Metric C that is calculated using the well understood metrics A and B. By simply using the metrics A and B, the knowledge about these two metrics can be used to determine the feature that C measures. Furthermore, the calculation of C is easier to understand if two well known metrics are used, instead of some complex rule. Additionally, it shows that there is a relationship between C and the metrics A and B. If C is calculated directly, it is not clear that there is a between these metrics.

Name	**Description**	**Allowed operations**
nominal scale	unordered set of classification for attributes	$=, \neq$
ordinal scale	ordered set of classification for attributes, without a known distance between the classifications	$=, \neq, <, >$
interval scale	totally ordered set of numerical classifications with an arbitrary offset	$=, \neq, <, >$, distance between measured values
ratio scale	totally ordered set of numerical classifications	$=, \neq, <, >$, distance between measured values, addition, multiplication, division
absolute scale	same as the ration scale, but it does not make sense to convert the scale proportionally using division and multiplication	$=, \neq, <, >$, distance between measured values, addition, multiplication

Table 2.1.: A classification of measurement scales

Now that software metrics have been introduced in general some details have to be considered. Up to now, nothing about the output (or the result) of a software metric has been said. The reason is that this is not as simple as it seems. There are different kinds of scales that a metric can map its results to. A possible classification of the different metric scales can be found in table 2.1. For each kind of scale different operations are allowed, depending on the scale itself. The reason that not every mathematical operation is allowed lies in the scales themselves. If one considers an ordinal scale, it is obviously not possible to divide the measured values by some number, because the metric values themselves are not numerical.

The metrics that are used in this thesis are using a ratio scale. This is important because only ratio scales allow to change the scales using division. Furthermore, the metrics can be separated into categories by the general attributes they measure. Two of these categories are the process and the product metrics. The process metrics measure the process of software development directly. Attributes measured by a process metric are for example the number of failures, the amount of working hours spent on the project or the amount of money that the project costs. A product metric measures the product that is developed directly.

Now that the basics about software metrics are established two software metrics will be introduced in detail. More metrics will be introduced together with the tools that are used to collect them in chapter 3.

2.1.1. The software metric Lines of Code (LOC)

This metric does exactly what the name says: it counts the lines of code of a software project. Because it is evaluated using the source code, it is a product metric. This simple metric is probably the most popular software metric. However, there is a problem. The name does not say how empty lines or comments are handled. Thus, when it comes to lines of code, one has to define exactly which lines are counted. Here are some definitions for Lines of Code that are often used:

- Total Lines of Code (TLOC): Counts every line, including lines that only contain comments or are empty.
- Non-empty Lines of Code (NELOC): Counts only non-empty lines, including lines that contain only comments.
- Non-commented Lines of Code (NCLOC): Counts only none empty lines and lines that do not contain only comments.
- Delivered Lines of Code (DLOC): Delivered lines of Code, this excludes test code or assertions.
- Commented Lines of Code (CLOC): Counts only lines that contain a comment.

These are only some examples of possible definitions for the Lines of Code. In this thesis, the definition of TLOC is used. So every time that LOC is used, TLOC is meant.

2.1.2. The software metric Number of Bugs (BUG)

When it comes to analyzing the quality of a software project, there are usually two things that managers and users consider. The first is, if the requirements have been met and the second is how many failures there are. Failures are produced by bugs. Thus, the metric BUG measures an important attribute concerning the quality of a project. It belongs to the process metrics, because it analyzes not the product itself, but the result of the process. There are several problems with this metric. The first is how it can be measured. There is no way to directly know all the bugs that are in a piece of software. However, the known bugs can be used to estimate the total number of bugs. Usually the known bugs are documented in some form, so this metric can be measured using the documentation. The general problem of this metric is that humans directly influence it. Only the known bugs are counted, and which bugs are known depends on the persons that try to find bugs. Thus, the value might not be the same if there were other users. More details about this metric will be introduced in section 3.3, when a way to measure this metric is shown.

These basic concepts about software engineering are sufficient to understand the reasoning behind the performed experiments and the terms that are used.

2.2. Machine Learning

This section is similar to the Software Engineering section. First the question has to be answered what the abstract term of machine learning means. Again, it is difficult to answer because machine learning is such a wide field. After a first general definition of machine learning according to [14], some examples are presented to gain a better understanding.

Definition 2.9. Machine Learning
A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Because this definition is very abstract, consider the learning of a rectangle in the 2-dimensional real-space as an example. The learning problem is the following: Given is a *training set* of points $D \subset \mathbb{R}^2$, such that for each point it is known whether it is inside the rectangle R or not. This is a binary classification. It can be seen as tuples $< x, f(x) >$ such that $f(x)$ is 1, if $x \in R$ and 0 otherwise. The rectangle R itself is unknown. The aim of the learning problem is it to estimate the rectangle R. One algorithm to solve this problem is to compute the smallest rectangle R', such that for all $x \in D$ the classification from above holds: if $f(x) = 1$, $x \in R'$ and if $f(x) = 0$, $x \notin R'$. An example for this learning task and the solution of the algorithm is shown in figure 2.1.

This algorithm is a learning algorithm according to definition 2.9. The experience E are the tuples $< x, f(x) >$, the task T is to estimate the rectangle. The performance measure P is the error of the estimated rectangle R'. How the error could be defined will be discussed later on. To learn the classification of something is a common learning task that can be applied in many fields. The question usually is something like “is my product good or bad”. Thus, the *samples* x with $f(x) = 1$ are called *positive* examples, the others are called *negative* examples. This kind of binary classification is also known as discrimination.

The type of learning algorithm described above knows the classification of the data. This type of data is called *supervised* data. The other general kind of data is the *unsupervised* data. For unsupervised data, the classification is not known. Accordingly, algorithms are called supervised or unsupervised depending on the kind of data they need as input. Examples for unsupervised learning algorithms are clustering algorithms, which are described in more detail in section 2.2.1.

Now that a general outline of learning problems is clear, the structure of learning problems will be defined in greater detail. Let $\mathcal{F}$ a class of concepts over an input space X and let Y a target space. A class of concepts is a class of functions $f : X \to Y$. The functions $f \in \mathcal{F}$ are called concepts. The hypothesis space $\mathcal{H}$ is defined in an analogous manner. In the rectangle example, the concept space and the hypothesis space were both the rectangles over the 2-dimensional real-space. However, generally the concept space and the hypothesis space are not the same. In most practical learning problems, the concept space is unknown. Consider

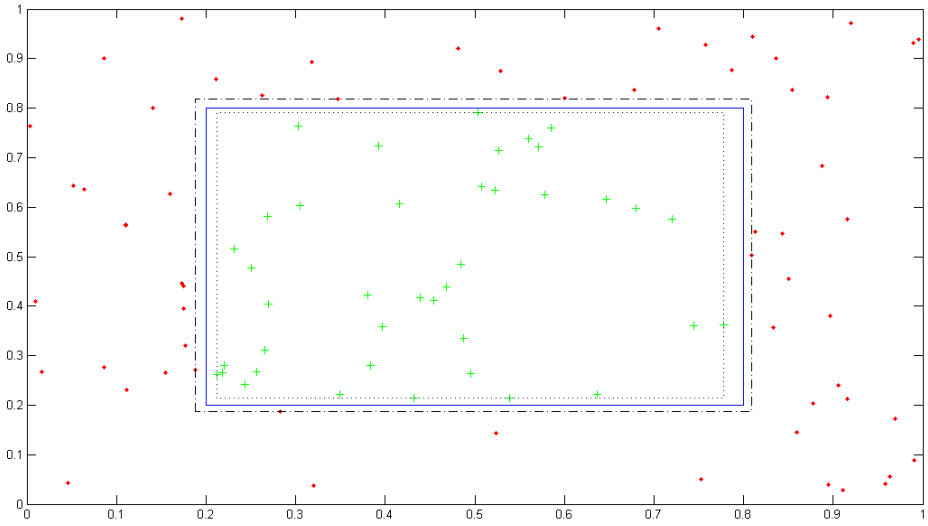

Figure 2.1.: The red dots are the negative examples; The green crosses the positive examples; The blue rectangle is the original rectangle, used to classify the data; The dotted line is the smallest rectangle, such that all positive examples are inside; The slash-dotted line is the largest rectangle, such that all negative examples are outside.

for example the field of gen analysis using machine learning methods. It is not known how the structure of genes is generated, but it is possible to predict certain features using models. In this thesis the input space can be considered as the n-dimensional real-space with metric values as input. The dimension n is equal to the number of metrics used for the learning task.

In practical applications, a general problem with the observed input data is *noise*. This means, that the observed data deviates from real data. There are different kinds of noise and some are better to handle than others. There are many possible reasons for this inaccuracy, for example human error or physical restrictions. Sometimes the complexity of the feature that shall be observed is the problem. Whatever the reason for the noise is, the problem is the same: if the input data is inaccurate, it is only logical that the output will be inaccurate to some degree. Furthermore, many learning algorithms do not work in the presence of noise. The introductory example of the rectangle learning is one of those algorithms. Only one wrong classified point can be sufficient, such that no rectangle separating the positive and the negative classified points exists any more.

The output space depends on the kind of learning that takes place. In this thesis two different kinds of learning algorithm are used: in the main part of this thesis a clustering algorithm, as it is described in section 2.2.1, is used. This kind of algorithm learns a classification of the

data. The common cases of classification are the binary classification where the output space is $\{0, 1\}$ and the classification in k classes, where the output space is $\{0, \ldots, k-1\}$. Another approach is not to learn the classification of data, but a function that describes the data and extrapolates regions where no data is available. To demonstrate how function learning could be applied to the measured data the k-nearest neighbour algorithm will be introduced in section 2.2.2.

If the different kinds of learning methods described above are considered, it becomes clear that different measures for the error are needed for the different learning tasks. In the following, the different types of errors that are needed to evaluate the algorithms used in this work are introduced, formally defined and, if possible, compared.

Definition 2.10. True Error
Let $f : \mathcal{X} \to \{0, 1\}$ a concept from a concept space $\mathcal{F}$ and $h : \mathcal{X} \to \{0, 1\}$ a hypothesis from a hypothesis space $\mathcal{H}$. Let $\mathcal{D}$ a distribution over $\mathcal{X}$. The true error of h in respect to f is defined as

$$error_{\mathcal{D}}(h) = \mathbf{Pr}_{x \in \mathcal{D}}[f(x) \neq h(x)]. \tag{2.3}$$

In the above definition $x \in \mathcal{D}$ means that $x \in \mathcal{X}$ is drawn according to the distribution $\mathcal{D}$. The problem with this definition is that one has to know two things that are usually unknown in practical experiments to calculate the true error: The true concept f and the true distribution $\mathcal{D}$. Even if the distribution is sometimes known, the true concept f is still unknown. Otherwise there would be no need to learn a hypothesis, unless the concept is too complex to calculate it. However, the difference between an uncalculable concept and an unknown concept is not important – in both cases they cannot be used for the computation of the true error. The following definition is similar to the definition of the true error, but it only needs a sample to calculate the error.

Definition 2.11. Sample Error
Let $f : \mathcal{X} \to \{0, 1\}$ a concept from a concept space $\mathcal{F}$ and $h : \mathcal{X} \to \{0, 1\}$ a hypothesis from a hypothesis space $\mathcal{H}$. Let $\mathcal{D}$ a distribution over $\mathcal{X}$. Let $X \subseteq \mathcal{X}$ a finite subset of $\mathcal{X}$.

The sample error of h in respect to f and the sample X is defined as

$$error_S(h) = \frac{1}{|X|} \sum_{x \in X} \delta(f(x), h(x)) \tag{2.4}$$

with $\delta(f(x), h(x)) = \begin{cases} 1 & \text{if } f(x) \neq h(x) \\ 0 & \text{otherwise} \end{cases}$.

Using the definition of the sample error, it is possible to determine the error of a hypothesis in reference to a subset of the input space where the values of the concept are known. This is the case for the training data in every supervised learning algorithm. Usually the training data is split to evaluate the error correctly. A part of the data is used as a training sample for

the learning algorithm to calculate the hypothesis. Another part is used to evaluate the error of the calculated hypothesis. The reason for this is simple: Most learning algorithms minimize the error of the calculated hypothesis on the training sample. However, it is not known if the hypothesis is good on another sample. By splitting the data, the error of the hypothesis can be calculated on a sample that is independent of the training data.

A drawback of the definition of the sample error is the usage of the δ function. An interesting feature of a learning algorithm is also what kind of error it produces. As an example, consider a learning algorithm that shall determine whether someone is credit worthy or not. A false positive result of learning algorithm can result in a big loss for the bank. On the other hand, a false negative is no big loss. The other way around is often the case too. When testing for illnesses, a false negative is often not an option. For example, if an AIDS test were negative, but the person would actually have AIDS it would be a serious problem. The following definition of the error introduces a measure for false positives and false negatives.

Definition 2.12. Error of the first and second kind
Let $f : \mathcal{X} \to \{0, 1\}$ a concept from a concept space $\mathcal{F}$ and $h : \mathcal{X} \to \{0, 1\}$ a hypothesis from a hypothesis space $\mathcal{H}$. Let $\mathcal{D}$ a distribution over $\mathcal{X}$.

- The error of the first kind is defined as

$$error^1_{\mathcal{D}}(h) = \mathbf{Pr}_{x \in \mathcal{D}}[f(x) = 0 \wedge h(x) = 1]. \tag{2.5}$$

- The error of the second kind is defined as

$$error^2_{\mathcal{D}}(h) = \mathbf{Pr}_{x \in \mathcal{D}}[f(x) = 1 \wedge h(x) = 0]. \tag{2.6}$$

Thus, the error of the first kind measures the probability of a false positive, while the error of the second kind measures the probability of a false negative. By decreasing the error of the first kind, usually the error of the second kind is increased and vice versa.

As an example, consider two strategies for solving the rectangle learning problem. The first strategy is to calculate the smallest rectangle, such that all positive examples are inside the rectangle. The second strategy is to calculate the largest rectangle, such that all negative examples are outside the rectangle. As it can be seen in figure 2.1, the first strategy calculates a smaller rectangle than the second stragegy. The space between both rectangles will be classified differently by both strategies. While the result of the first strategy will classify the area between itself and the real rectangle as false negative, the second strategy will classify the area between itself and the real rectangle as false positive. As can be seen, while the error of both hypothesis is nearly the same, the error of the first and second kind is different.

All of the above definitions of error can only be applied to classification learning algorithms. If you consider the functions $f_1(x) = x$ and $f_2(x) = x + \varepsilon$ for an $\varepsilon > 0$, the above definitions of the true error or sample error of f_2 in reference to f_1 would be 1. However, for a small ε,

f_2 would really be a very good approximation of f_1. Of course this is an artificial example, but it should clarify the need for a totally different definition of the error for function learning. One possible definition is the residual error.

Definition 2.13. Residual error
Let $f : \mathcal{X} \to \mathcal{Y}$ a function, $\hat{f} : \mathcal{X} \to \mathcal{Y}$ an approximation of f and $||\cdot||$ a norm over $\mathcal{Y}$.
The residual error of $\hat{f}$ with respect to f at a point $x \in \mathcal{X}$ is defined as

$$error(\hat{f}, x) = ||f(x) - \hat{f}(x)||. \tag{2.7}$$

The residual error does not measure the error of the complete hypothesis, but the error at a single point. This way it is possible to determine if a hypothesis is locally a good approximation of a function, but the global error can not be measured. However, it is possible to estimate the global error using the residual error in a way similar to the sample error. Given a training sample X, the error can be estimated as the maximum residual error $error(\hat{f}) = \max\{||f(x) - \hat{f}(x)|| : x \in X\}$. The above definitions of error are sufficient for the further algorithms and the analysis of the results in the later parts of the thesis.

The Bayes theorem is an important theorem in the theory of machine learning. It serves as basis for many learning algorithms, for example some clustering algorithms. The statement of the theorem is basically that the probability of a hypothesis given a training sample can be calculated using only the probability of the training sample given the hypothesis, the probability of the training sample given the distribution of the data and the probability of the hypothesis itself.

Theorem 2.14. *Let $h \in H$ be an hypothesis from some hypothesis space H and D the observed training data. Then the probability of the h given D can be written as*

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}. \tag{2.8}$$

This does not seem like it simplyfies anything. In most practical problems the distribution is unknown. Thus, one has to assume that the distribution of the training sample is the same as the real distribution. So the probability of the training sample given the distribution is not a problem. As for the probability of the hypotheses, it is usually a mistake to make assumptions about the probability of a hypothesis. In this case, all hypothesis should have the same probability. Otherwise there is a risk that wrong a-priori knowledge is used, which will usually lead to bad results. This only leaves the probability of the observed training data given the hypothesis. However, for a given hypothesis this can be calculated. Altogether, given only the hypothesis and a training sample it is possible to determine how probable the hypothesis is for the observed data. However, there are some drawbacks as well. If the distribution of the training data is different to the real distribution, this will falsify the classification. So as one expects, everything depends on the quality of the training data.

It would be too much to discuss the Bayes theorem in greater detail. On the other hand, the Bayes theorem serves as the basis for the learning algorithms used, so it would not have been right to leave it out all together. Interested readers should take a look at the literature mentioned at the begin of this section. The basics given here can be seen as the tip of the iceberg when it comes to machine learning.

2.2.1. The k-means clustering algorithm

In many applications training data is available, but the sources that created the data are unknown. These data sources can be estimated by clustering algorithms. After estimating the sources, each point of the training sample is associated with the source that probably generated it. The kind of the source depends on the clustering algorithm. Examples for possible sources are probability distributions or classification problems. In our case, the sources are different classes which the values originate from. Thus, the assignment of a source to a training sample can be considered as a kind of classification, where the data sources are the classes. Since the classes are unknown, the cluster algorithms use a heuristic to assign the data points to the sources. Usually these heuristics use some kind of *distance* between the data points to assign them to their sources. Because of this, the sources are called *clusters* in this context. A cluster consists usually of points with a close proximity, so the points inside a cluster are literally clustered. As clustering algorithms only use the characteristics of the data itself, but no classification or function values, they belong to the unsupervised learning algorithms.

The output of every cluster algorithm is the same: A map that assigns each data point to its cluster. But there are several questions that have to be answered beforehand. The most important one is, how many sources and thus clusters are there. Basically, there are two kinds of clustering algorithms: those that calculate a fixed number of clusters, where the user defines the number; those that determine the number of clusters themselves. Here the k-means algorithm is used. It belongs to the first type and k is the number of clusters. It is a very simple and very popular clustering algorithm.

The k-means algorithm calculates k clusters from a finite training sample $D \subseteq \mathbb{R}^n$. Each cluster is defined by a center $c_i \in \mathbb{R}^n$ for $i = 1, \ldots, k$. A data point $x \in \mathbb{R}^n$ is part of cluster i, if i is a solution of the optimization problem

$$\min\{||c_i - x||_2 : i = 1, \ldots, k\}. \tag{2.9}$$

If there is more than one solution, the smallest i is chosen as cluster. Since the training data $D \subseteq \mathbb{R}^n$ this can be used to determine to which cluster each vector of the training data belongs.

The algorithm computes the centers in iterative way. Initially the centers c_i are arbitrary values. They could either be defined by the user using some a-priori knowledge or simply be random values. In each step of the algorithm the current clusters $C_i \subseteq D$ are computed using the rule defined by equation 2.9. Afterwards the cluster centers are updated. They are set to the mean

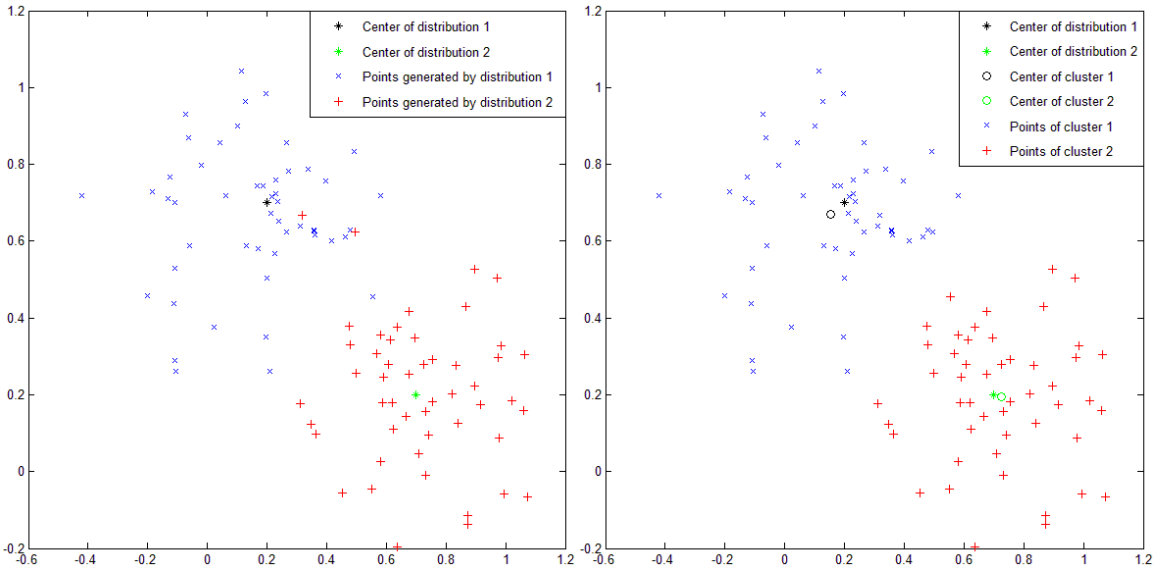

Figure 2.2.: On the left: simulated data of two different normal distributed sources represented by blue and red crosses in the plane. On the right: clusters found by the k-means cluster algorithm.

value of all points that belong to the cluster:

$$c_i = \frac{\sum_{x \in C_i} x}{|C_i|} \text{ for all } i =, 1 \ldots, k. \tag{2.10}$$

These steps are repeated until convergence is reached. By choosing the cluster centers as the mean values of the elements of the cluster, the intra-cluster variance is minimized.

The result of the algorithm – that means the calculated centers – depend on the initial centers. Different initial centers can lead to different results. However, if the data sources generate clearly separated data, the result should be stable, that means they should for most, if not all, start clusters lead to the same result. Another problem that can occur is that of empty clusters. It is possible that as part of the iteration, one ore more clusters become empty. If this is the case, there has to be a strategy to handle this situation. There are two common possibilities to solve this problem. Either the empty cluster is removed and the result has less than k clusters or the algorithm terminates prematurely with an error. If different start clusters are chosen, it is possible that the algorithm terminates correctly.

As for the error of the k-means algorithm, the error definitions for classification learning algorithms can be used. That means the true error and sample error can be measured if the true sources of the data are available. If there are $k = 2$ clusters and one can be seen as positive and one as negative, it is also possible to analyze the error of the first and second kind.

Algorithm 1: The k-means algorithm

Input : Number of clusters k, training sample $D \subset \mathbb{R}^n$, initial centers c_i^0 for $i = 1, \ldots, k$
Output: Calculated centers c_i^j for $i = 1, \ldots, k$
Init: $j \leftarrow 0$;
Start: Determine the clusters C_i^j by solving the optimization problem $\min\{||c_i^j - x||_2 : i = 1, \ldots, k\}$ for each $x \in D$.
For all $i = 1, \ldots, k$ update the cluster centers:

$$c_i^{j+1} \leftarrow \frac{\sum_{x \in C_i^j} x}{|C_i^j|} \tag{2.11}$$

$j \leftarrow j + 1$
if $c_i^j = c_i^{j-1}$ *for all* $i = 1, \ldots, k$ **then**
    return *Cluster centers* c_i^j *for* $i = 1, \ldots, k$
else
    Go to **Start** and repeat the above steps.
end

2.2.2. The Nearest Neighbour Algorithm

The other algorithm used in this work is the nearest neighbour algorithm. It is a supervised *instance-based* function learning algorithm. Instance based means that it will not compute a hypothesis, which can be used to evaluate data from the input space. Instead it uses a point of data – an instance – as input and evaluates the result itself. When it comes to the run-time, this can be a problem, because there is no "simple" hypothesis h as the result of running the algorithm once, which can then be used to evaluate a sample y from the input space to the value $h(y)$. The whole algorithm has to be run for every value that should be evaluated. This is usually a more complex operation compared to evaluating a hypothesis.

As input, the nearest neighbour algorithm uses supervised and finite set training data D from the n-dimensional real-space and for each element of the training set its value under some real-valued function $f : \mathbb{R}^n \to \mathbb{R}$. As the name suggests, the nearest neighbour algorithm uses the training data that is closest to a vector $x \in \mathbb{R}^n$ to estimate its value $\hat{f}(x)$. To determine the distance between two points, usually the euclidian distance

$$dist(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{2.12}$$

is used. However, it is possible to use other metrics to measure the distance, but this is not needed for this work. The reasoning behind using the nearest neighbours to estimate the value

of a vector under a function is that if the function is smooth, the value should be similar to that of its neighbours.

For a vector $y \in \mathbb{R}^n$, the simple nearest neighbour algorithm outputs the value under of f the vector in $x \in D$ closest to y, thus

$$\hat{f}(y) = f(argmin_{x \in D} dist(x, y)) \tag{2.13}$$

A generalization is not only to consider the nearest neighbour but the k-nearest neighbours $x_1, \ldots, x_k$ of y, for $k \in \mathbb{N}, k \leq |D|$.

Using k neighbours has the advantage that the reliance and the smoothness of the function is reduced. Consider the case that the nearest neighbour $x^{'}$ is an outlier. If only that value would be used, y would be assigned the value of the outlier too, which would lead to a large residual error. If not only the nearest neighbour, but other neighbours are taken into consideration, this effect can be reduced. The same argument holds for noisy data. By using more than one value from the training data for the calculation, the negative effects of the noise can be reduced. However, if k is too large, there is a chance that vectors from "far away" are part of the k-nearest neighbours. This would destroy the concept to use points that are close.

If the approximation $\hat{f}(y)$ is calculated using k values from D, the used values can be weighted with different weights w_i. Usually two possibilities for weights are considered:

1. All neighbours are weighted equally, thus $w_i = 1$ for all $i = 1, \ldots, k$.
2. The neighbours are weighted by their squared distance from y and $w_i = \frac{1}{dist(x_i, y)^2}$ for all $i = 1, \ldots, k$.

Both cases have their advantages. If the neighbours are equally weighted, an outlier close to y will be better compensated. On the other hand, if the values close to y are no outliers, it is advantageous to weight the closer neighbours heavier. Through using the squared distance, vectors with a larger distance are heavily penalized. This is a trade-off between both advantages – reducing the effects of outliers and still give greater weight to the closer points – hoping to gain both of them. In the second approach it is also possible that the denomiator is zero, if $y \in D$. The simple solution to this problem is to return the already known value, thus $\hat{f}(y) = f(y)$. In practical applications this is often not that simple. If the data is noisy it is possible that several values for some vector $x \in \mathbb{R}^n$ are available. In this case, there are two options. Either the several values are reduced to only one value using precomputation or the nearest neighbour algorithm (Algorithm 2) has to be adapted. In the else-branch the set $W_0 \subseteq \{i : w_i = 0$ and $i = 1, \ldots, k\}$ has to be determined an equation 2.16 has to be replaced with

$$\hat{f}(y) = \frac{\sum_{i \in W_0} f(x_i)}{|W_0|} \tag{2.14}$$

Algorithm 2: Weighted k-nearest neighbour algorithm

Input : Training Set D with values $f(x)$ for all $x \in D$, $y \in \mathbb{R}^n$
Output: Estimated value $\hat{f}(y)$
Determine $\{x_1, \ldots, x_k\} \subset D$, such that the x_i are the k closest elements to y in D.
if $w_i \neq 0$ *for all* $i = 1, \ldots, k$ **then**

$$\hat{f}(y) \leftarrow \frac{\sum_{i=1}^{k} w_i \cdot f(x_i)}{\sum_{i=1}^{k} w_i} \tag{2.15}$$

else
Determine i such that $w_i = 0$.

$$\hat{f}(y) \leftarrow f(x_i) \tag{2.16}$$

end
return $\hat{f}(y)$

2.2.3. Relationship between the k-means and the nearest neighbour algorithm

If one compares the strategy of the cluster assignment of the k-means algorithm with the way the simple nearest neighbour algorithm approximates the value $\hat{f}(x)$ the similarity is obvious. The similarity between the two algorithms is one of the reasons the nearest neighbour algorithm is introduced. It can be used to visualize the results of the k-means algorithm. Let the set of centers $C = \{c_1, \ldots, c_k\}$ computed by the k-means algorithm and $f(c_i) = i$ for $i = 1, \ldots, k$. If this is used as input for the simple nearest neighbour algorithm, it yields the index of the cluster it would be assigned to, for each $x \in \mathbb{R}^n$. If a color is assigned to each of these numerical values, this can be used to visualize the areas of the clusters computed by the k-means algorithm. Such a coloring of the space is called Voronoi diagram. An example for a Voronoi diagram is shown in figure 2.3. This technique to color regions using only a central point and then determine the colors using the nearest neighbour algorithm is often used in graphical programming.

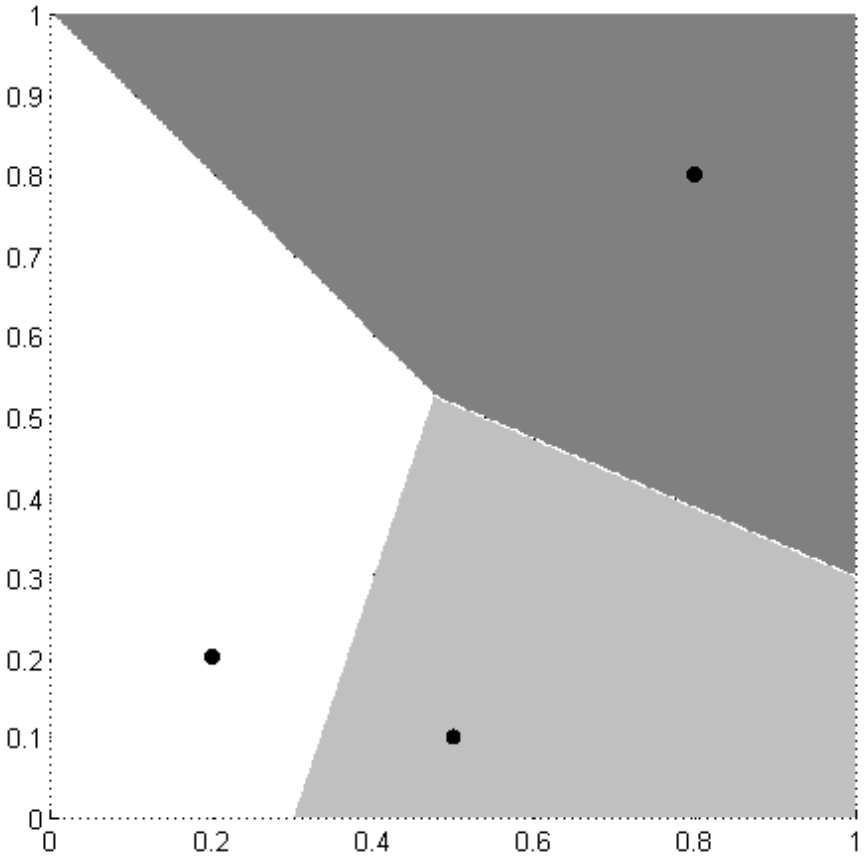

Figure 2.3.: This figure shows a Voronoi diagram that visualizes three clusters. Each color defines the area of one cluster, the black dots mark the centers of the clusters. It has been created using the nearest neighbour algorithm, with three vectors from the $\mathbb{R}^2$ as centers with the assigned values 0, 1 and 2.

3. Mining Data

In this chapter, the mining of the metric data is described in detail. There are several ways to mine data from a software project. The aim is to collect software metrics measured for significant versions of a project. As it was said in section 2.1, a software metric describes a feature of a complex software project as a number. The significant metrics for this work that are collected are the metrics BUG and LOC which were introduced in chapter 2. To collect LOC, the Java source code of the projects is measured using the Eclipse Metrics Plug-in; BUG is extracted from the Bugzilla bugtracking system of the Eclipse Foundation. To collect metrics for significant versions of a project knowledge about the project plan is required. Otherwise it is not possible to determine which versions were important. Important versions are usually defined in the project plan as Milestones, Alpha- and Beta-Versions or Release Candidates. These important versions are milestones after definition 2.2.

This chapter is divided into 3 parts: The first part describes how to extract the source code of specific version of a project from a repository of a source code versioning system. In the second part collecting of source code metrics, such as LOC, is discussed. The third part discusses how to extract data from a bugtracking system.

3.1. How to extract source code of specific versions from a repository

A compulsory requirement to obtain source code from different versions of a project is that the source code is still available. This is the case if the project uses a code versioning system such as CVS or SVN. However, not all revisions of such a repository are important for a project. Many revisions will not even contain consistent project snapshots. Thus it has to be determined which revisions of the repository contain important versions of a project.

Fortunately, code versioning systems have a so called *tagging* mechanism. This makes it possible to *tag* a revision of particular importance so that it can be easily checked out afterwards. If tags for the versions that shall be mined are available, the tags can be directly used to obtain the source code. There are two cases: the tag has the name of the version; the name of the tag associated with a version is stated explicitly in the project plan. Otherwise the revision of a version has to be determined using the project plan. Either the revision of the repository of a specific version is explicitly stated in the project plan or at least the date of the version is

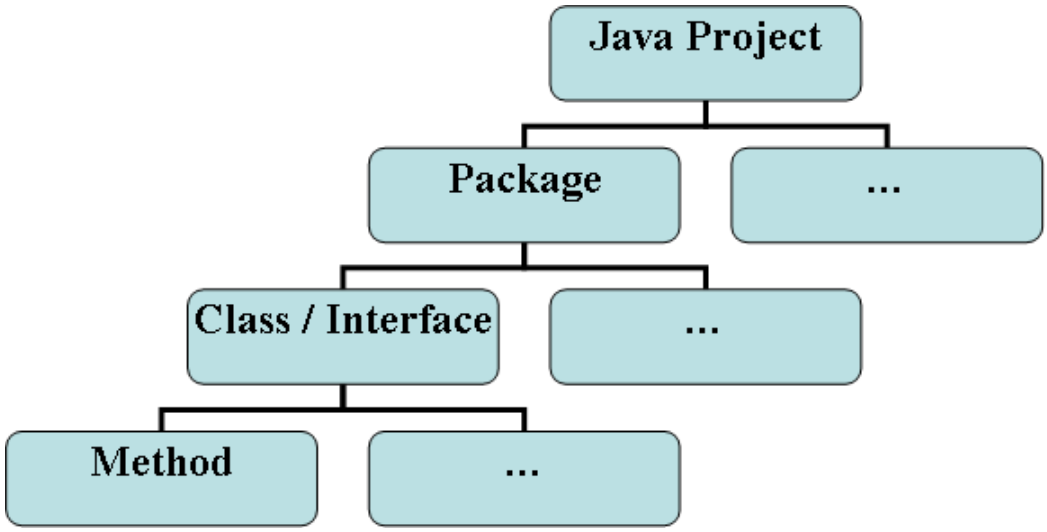

Figure 3.1.: The hierarchy inside a Java project: methods belong to classes, which belong to packages, which are part of a project.

available. Using the date, it is possible to determine the revision manually. If there is no revision at the stated date, latest revision before the date can be used instead.

After the tags, respectively the revisions to all the versions are determined, the source code can be obtained by checking out the associated content. Afterwards it can be used for further analysis.

3.2. Measuring Java source code

If the source code of a Java project is available, there are plenty of ways to measure it. For this thesis, the tool Eclipse Metrics Plug-in [3] was used to collect metrics. A metric that is collected from a Java project can be gathered at different *levels*. The level can either be a method, a class, a package or the whole project. The levels define a tree-like structure: a method is part of a class, a class is part of a package and a package is part of project (see figure 3.2). All of these metrics have different methods of measurement. As part of the mining for this thesis, all metrics that the Eclipse Metrics Plug-in can measure were measured and the results were exported.

To start a measurement with the Eclipse Metrics Plug-in all that is needed is a compilable Eclipse Java project. The metrics that the plug-in can measure are listed and described in table 3.1. The plug-in will collect all selected metrics automatically after the project is built. After the measurement is complete, the results can be exported to a XML file. Even though for the

experiments that were performed as part of this thesis, only the source code metric TLOC was needed, all available metrics were measured. The reason for this is quite simple: Maybe the other data will be needed for further experiments and it would be a lot more work to reiterate all the measurement steps again in comparison to just collecting all metrics at once.

Table 3.1.: A listing of the Java source code metrics measured using the Eclipse Metrics Plug-in. The first column describes the acronym which is used to identify the metric; The second column is the name of the metric; The third column holds the information at which level of the Java type hierarchy the metric is measured; The fourth column describes how the metric is measured.

Acronym	Name	Level	Description
CA	Afferent Coupling	Package	Number of classes used outside the package that use the package
CE	Efferent Coupling	Package	Number of classes inside the package that depend on other packages
DIT	Depth of Inheritance Tree	Type	Distance of the type from the Object class
LCOM	Lack of Cohesion in Methods	Type	A measure for the cohesion of a class, see [10]
MLOC	Method Lines of Code	Method	Lines of code of a method without comments
NBD	Nested Block Depth	Method	Nested block depth of a method
NOF	Number of Attributes (Fields)	Type	Number of attributes of a class
NOI	Number of Interfaces	Type	Number of interfaces a type implements
NOM	Number of Methods	Type	Number of methods of a type
NOP	Number of Packages	Project	Number of packages in a component/project
NORM	Number of Overridden Methods	Type	Number of derived methods that are overridden
NSC	Number of Children	Type	Number of types derived from this type

Continued on next page

Acronym	Name	Level	Description
NSF	Number of Static Attributes	Type	Number of static attributes of a type
NSM	Number of Static Methods	Type	Number of static methods of a type
PAR	Number of Parameters	Method	Number of parameters of a method
RMA	Abstractness	Package	Number of abstract classes and interfaces devided by the total number of type
RMD	Normalized Distance	Package	Indicator for the balance of abstract and concrete classes ($\|RMA + RMI - 1\|$)
RMS	Instability	Package	Compares the usage of the package from outside the package to the usage of other packages ($\frac{CA}{CE+CA}$)
SIX	Specialization Index	Type	Index for the specialization of a type compared to its parents ($\frac{NORM*DIT}{NOM}$)
TLOC	Total Lines of Code	Project	Total lines of code inclusive comments
VG	McCabes Cyclomatic Complexity	Method	Cyclomatic number of the control flow graph of a method
WMC	Weighted Methods per Class	Type	Sum of all McCabe complexety numbers for all methods of the class

3.3. How to extract bug data from a bugtracking system

To collect data about the number of bugs bugtracking systems, such as Bugzilla, can be analyzed. For this thesis a Bugzilla [1] database was analyzed, thus the following part is in reference to Bugzilla. However, other bugtracking systems have similar traits, but sometimes the names are different. Bugzilla is used to gather and maintain all information about bugs. Additionally it can handle requests for enhancements. It is able to maintain several projects including their components at once.

A Bugzilla entry contains all information necessary about a bug. A description of the bug, the severity of the failure it causes, the current status, the version of the software it occurred in and if it has been fixed, also the version where it has been resolved. Table 3.3 lists these attributes in greater detail and also gives exemplary values. Usually there is more information than just the listed attributes available, but since they are not important for this work, they are not discussed here.

Name	Description	Typical values
Severity	Severity of the bug. Can also be enhancement if it is a request for an enhancement.	trivial, minor, normal, major, critical, blocker and enhancement
Status	Current status of the bug.	UNCONFIMERMED, NEW, ASSIGNED, RESOLVED, VERIFIED, CLOSED
Resolution	Resolution of the bug or no entry at all if there is no resolution yet.	FIXED, INVALID, WONTFIX, DUPLICATE
Reporter	The person that reported the bug.	Joe Everyone (joe.everyone@mail.com)
Assigned To	The person or group that is responsible for fixing the bug.	John Smith (john.smith@company.com)
Summary	A short summary of the problem that the bug causes.	The feature X does not work.
Description	A detailed description of the bug.	It does not work when you do something.

Table 3.2.: Description of typical bugtracking system entry information

To measure information about the number of bugs of a software project, the most interesting attributes are the version it has been reported/fixed in and the severity.

Using these two metrics, the following metrics can be calculated:

- The total number of open bugs. This is the same as the metric BUG that was introduced in section 2.1.2.
- The number of fixed bugs between two versions.
- The same as the two above, but separately for each severity.
- A weighted number of bugs, with their severity as weights.

To collect metrics from a Bugzilla database, there are two options. In case the underlying database is publicly accessible, either in form of a downloadable dump or directly via SQL queries, the metrics can be measured automatically. Otherwise the web interface has to be used. How the automated collection can be done will be discussed in chapter 4, where a tool to build a large database from all the mined data is introduced. If it is not possible to directly access the database, the information has to be extracted manually using the web interface of the Bugzilla database.

4. A tool to pre-process the mined data

There are several tasks that have to be performed on the available metric data that can not be done manually. Thus, a tool is needed to perform these tasks. They are mainly the collection of the measured metrics from the different sources and the preparation of the input for learning algorithms. Another task that has to be done is the extraction of the bug-metrics from the bugzilla database. As it was said, this can be done using the SQL database that is the backend of a bugtracking system. If a new tool has to be created to gather metric data from different sources into a single database, it can also be used to extract these metrics from the SQL database. The database is just another source of metric values.

Before the tool itself is introduced, one should take a look at the way the data is collected. Using the approach to mine data from software repositories that was introduced in chapter 3, the result will be many files that contain pieces of the mined data. Collecting software metrics using the Eclipse Metrics Plug-in will result in one XML file per Eclipse project that has been measured. Thus, there is at least one file for each measured version of a project. If a project consists of several sub-projects, there is more than one file per measured version.

It is clear that with a growing number of versions and sub-projects it is no longer possible to extract data manually. Another problem is that the sources of data do not even have the same structure. While the source code metrics are exported as XML files, the backend of a bugtracking system is usually a SQL database. If metric data is extracted manually, it can be stored in an arbitrary format. Furthermore, the structure of the input data that a learning algorithm requires is not clear. To analyse the mined data, it has to be possible to purposefully extract data from the complete set of mined data.

Altogether, the amount of data and the different input/output types of the data make it clear that a tool to handle theses tasks is needed. Additionally, the tool should be able to merge all of the input data into a single large database, thus reducing the fragmentation of the measured data.

In the following sections the requirements on the tool, some important design decisions and import details of the implementation are discussed.

4.1. Requirements

The most important requirement on the tool is that it has to be able to handle different kinds of input and output formats for the metric data. To simplify the conversion between different input

and output formats, internally only one format should be used to store all the data. This way, the import functions and export functions do not need to know anything about each other but only have to read/write the data from/to the common internal format. However, this format needs to be very flexible. It has to be flexible enough to satisfy all requirements that different external formats pose.

As it was already mentioned in section 3.2, metrics can be measured on different levels and these levels define a tree-like structure. The internal type needs to be able to store the data in such a tree-like structure. However, what levels are needed is not know beforehand. They can vary with different projects, for example because of the progamming language that is used. Thus, the levels can not be predefined except the highest level, the project itself. If the data were only stored in a *flat* format, a format where the levels are lost and only the values are retained, most of the measured information would be lost. For example, metrics collected from methods can be used to calculate an indirect metric for the class the methods belong to. If the data were stored in a flat format, this would not be possible anymore.

For each level it is neither known how many metric values need to be stored nor if there is a level below. For the level below it is also not known how many entities there are. This means that every node has to be able to store an arbitrary number of values and nothing about the number of children is known. However, for each measured value must still be a unique way to identify it available.

So altogether, the internal representation needs to fulfill the following requirements:

- A tree structure that represents the internal hierarchy of the project data.
- All nodes of the tree structure need to be able to store an arbitrary number of metric values.
- For all measured values it must still be possible to identify them.

As it was already said at the beginning of this chapter, the tool also needs to be able to access a SQL database used by a bugtracking system to extract metric data about the number of bugs from it automatically. To extend the support for more metrics and also for easier data preparation, indirect metrics are useful. However, as it was said in definition 2.8, every metric that can be calculated from other metrics is considered to be an indirect metric. It can not be expected that it is possible to calculate arbitrary indirect metrics. However, some important and rather simple ways to calculate indirect metrics should be supported:

- Normalization of a metric over all milestones. Further information about this is given in section 5.2.
- Creating a new metric as the sum of other metrics of the same level. For example, this can be used be used to calculate the total number of bugs from the number of bugs separated by their severity.

- Creating a new metric as the sum of all metric values of a metric in a sub-level. For example, this can be used to calculate a metric that measures the complexity of a class from the complexity of its methods.

Another requirement is that the tool should be able to perform a batch of operations according to a script. The reason for this is that the steps that have to be performed to read and write data are often similar. However, there are many small things that can vary. Here are some examples:

- The number, type and location of input. There can be data about one or several projects, the data could be in the form of XML files generated by the Eclipse Metrics Plug-in or an SQL dump of a bugtracking system.
- The type of the export. Depending on what is intended to do with the data after the export, different formats might be feasible.
- The metrics that shall be exported. For different learning tasks, different metrics may be required.
- The indirect metrics that need to be calculated. Depending on the task, different indirect metrics may be required.

If the tool was only able to perform complex operations that read from a specific kind of source, perform some operations on the data and than export it to some format it would be very inflexible. On the other hand, these complex operations can be split into smaller operations: read from a specific kind of source; perform an operation on the data; purposefully export data to a specific format. If these simple operations can be arranged in scripts, these scripts can easily replace the need for mighty operations.

Now there are four main requirements on the tool:

1. Read data from different kinds of data sources.
2. Common internal representation of the data.
3. Calculate indirect metrics.
4. Support scripting of operations.

4.2. Design decisions

The first design decision that was made was to implement the tool in Java. The advantage of Java is that with JDom [4] a mighty XML API exists and with JDBC SQL databases can be easily accessed. Using JDom allows easy access to the XML files that the Eclipse Metrics

Plug-in exports. By accessing an SQL database that is underlying a bugtracking system using JDBC, the metric BUG can be easily extracted in an automated way.

Because of the specific needs of the internal representation of the data, a new tree structure was programmed. The root contains all information about the project itself, such as the version of the name of the project. The subnodes and measured values are maintained using lists. The details about this structure are described in section 4.3.1.

For the user interface, a simple text console was chosen. However, the interface is designed using the observer pattern. This way it can easily be replaced by a graphical user interface. The commands that are performed on the data are handled by a command parser, which is also independent of the input. All it needs is a string that defines the command. The form of the strings is defined by the following EBNF like grammer:

```
<command> := <commandname>{<whitespace><parameter>}
<commandname> := Valid Java class name
<parameter> := <string>|<stringarray>
<string> := <stringwithoutwhitespaces>|<stringwithwhitespaces>
<stringwithoutwhitespaces> := a string without whitespaces
<stringwithwhitespaces> := a string in quotation marks
<string> := a string that can have whitespaces, but must be in double quotes
<stringarray> := "[" <string>{<whitespace><string>} "]"
```

The command parser then calls the commands using a combination of the command pattern and reflection. The commands themselves implement an interface that describes the methods a command has to provide. All commands are all classes inside the same package. The name of the class defines the command name, furthermore, they have CMD as name prefix. For example, to create a command “load" the class has to be called “CMDload". The parameters are parsed by the command parser and passed as a map to the called command. Because of this usage of reflection, a new command can be added by simply adding a new class. To allow scripting, a command was implemented that reads a text file and each line of the file is treated as a command and executed.

4.3. Implementation Details

4.3.1. The core package

The core functionality of the tool, that means the storage of metric data and the calculation of indirect metrics is implemented in the `de.ugoe.cs.swe.databasebuilder.core` package.

The most important classes of the core are those that store the data internally. To represent the entity over which the data was measured, a class `MetricDataSet` was created. As it was

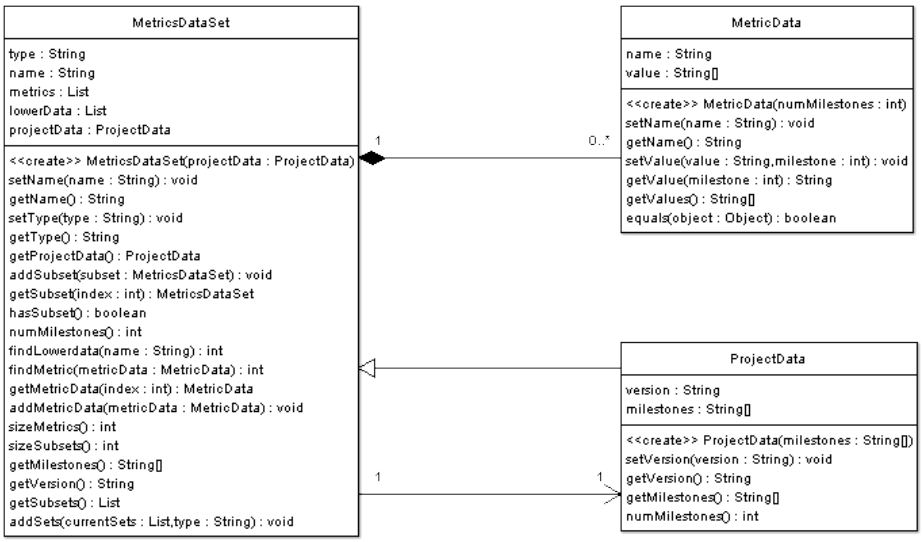

Figure 4.1.: This UML chart shows the classes used to store the metric data internally and their relationship.

said in the previous section, the data should be stored in a tree. The `MetricDataSet` class also represents the tree nodes. Thus, a `MetricDataSet` has to handle three things:

1. All the important information about itself, like the type and the name.
2. The metric data associated with itself.
3. Its children, that means entities that are one level deeper in the hierarchy.

A metric value can be identified by the triple of the entity it was measured over, the name of the metric and the version it belongs to. If every entity contains a list of its own metric values stored together with the names of the according metrics, all measured values can be identified. To store the metric values, the data class `MetricData` is used. It stores the name of the metric together with the metric values for each version. The values are stored in an array, thus it is not possible to change the number of versions afterwards it has to be known when an instance of `MetricData` is created. To which version a metric value belongs is implicitly given through the index in the array. Even though most metric values are numbers, the storage as a string is no problem because it can simply be solved by conversion. The `MetricDataSet` class contains a `java.util.List` of the metric data classes to store all the values. Because an entity can have more than one child - for example a package contains usually more than one class - the children of an entity are stored in a `java.util.List`, too.

The project is an entity of special importance. First of all, every entity that belongs to a project is a descendant of the project. Thus, the project is automatically the root of the entity-tree of itself. Additionally, a project has in comparison to the other entities a version, for example 1.0. Furthermore, the milestones that are measured are stored in the project entity. To realize the project as a special entity, a subclass `ProjectData` of `MetricDataSet` was created. To implement the differences between a project and the other entities, the class `ProjectData` has additional members. A string to store the version of the project and a string array to store the names of the measured versions. The list of metric values that are stored in all instances of `MetricData` have to be aligned according to names of of the measured versions. To give all entities of a project access to this information, each `MetricDataSet` stores a reference to the project that it belongs to and thus the root of the tree it belongs to. Figure 4.2 visualizes the relationship between these three classes.

The last part of the core is the class `MetricCalculator`. This class provides several member functions that allow the calculation of indirect metrics. All commands (see 4.4) are done using this metric calculator and the calculations are discussed together with the commands.

4.3.2. The data import package

Now that the internal storage and basic data handling is discussed, the question of how the data is loaded to this internal data format has to be answered. The classes that address the extraction of data from different sources and store the data inside the tool are organized in the `de.ugoe.cs.swe.databasebuilder.core.importdata` package. In the current version of the tool there are 3 ways to import data: to read data that was measured using the Eclipse Metrics Plug-in; to extract data from the underlying database of a Bugzilla bugtracking system; to read a database, that was created by the tool itself.

Importing data measured with the Eclipse Metrics Plug-in

An important task of the tool is to import data that was measured using the Eclipse Metrics Plug-in. As it was said in 3.1, the plug-in stores the measured values in XML file. A short example of such an XML file is shown in listing 4.1. This XML file is ordered by the metrics, the type of the entity over which the metric is measured is given as the `per` attribute of the `values` node. Table 4.3.2 shows to which Java types the values of `per` refer.

```
<?xml version="1.0" encoding="UTF-8"?>
<Metrics scope="org.eclipse.jdt.launching" type="Project" date="
    2008-05-14" xmlns="http://metrics.sourceforge.net/2003/Metrics-
    First-Flat">
    <Metric id = "NORM" description ="Number of Overridden Methods">
        <Values per = "type" total = "61" avg = "0,884" stddev = "1,44"
            max = "8">
```

```
        <Value name="Standard11xVMType" source ="Standard11xVMType.
            java" package ="org.eclipse.jdt.internal.launching" value
             ="8"/>
      </Values>
   </Metric>
</Metrics>
```

Listing 4.1: A simplified example of an XML export of metric data measured using the Eclipse Metrics Plug-in. A real export contains more metric and value nodes and for each metric more values.

As it was said in section 3.2, for each Eclipse project that is measured one XML file is generated. A large software project may consist of several components and each component may be split into subcomponents. In this case, there is an Eclipse project for each subcomponent. Thus, to measure the software project many Eclipse projects have to be measured separatly and many XML files will be generated. Furthermore, these files exist for each measured version. The question that has to be answered is how the XML files are organized in a – preferably simple – way, such that the structure of how the components and subcomponents are organized is still maintained and the versions over which the measurement took place are separated.

The approach used for this work is quite simple. It simply uses folders. A folder structure of the following kind is used to organize the XML files:

"ProjectName/ProjectVersion/ComponentName/SubcomponentName"

The project folder can contain an arbitrary number of components, the component folder an arbitrary number of subcomponents. The XML files are named "version.XML" and placed in the folders of the subcomponents. The "ProjectVersion" has nothing to do with the measured versions of the development process. Instead it is the name of the final version, thus the version that has been developed, such as 1.0 or 2.0. This way all information about the project hierarchy and the measured versions is maintained and the storage is simple because it only uses the basic mechanism of file systems. In the case that a project does not have components or subcomponents, one simply has to create a folder with a name like "main" as component folder, the name of the subcomponent folder could for example be the name of the Eclipse project.

Eclipse Metrics Plug-in type	Associated Java type
packageFragment	A Java package. It is called package fragment because other projects could implement other parts of the package.
type	A Java type, such as a class, an interface or an enumeration.
method	A method of a Java type.

Table 4.1.: The left column shows the types as they are used by the Eclipse Metrics Plug-in, the right column the Java types they refer to.

To read the data, the folder structure has to be evaluated and the `MetricDataSet` and the `ProjectData` instances have to be generated. The subcomponents are ommitted in this structure, such that all the data of the subcomponents is merged into the component data. The XML files are read and parsed using JDom. Again, the instances of `MetricDataSet` for each measured Java entity have to be generated and organized properly. As a result, to read the data about a whole complex project that consists of many components and subcomponents only the folder structure as introduced above has to be created and the XML files have to be named like the versions.

However, there are some things that have to be considered. It is possible that a new component or subcomponent is introduced during the development process. For this component/subcomponent only data about the versions after it has been created are available. The reading function has to be able to deal with these possible empty versions correctly. This also poses another problem. As it was said, the number and the names of the measured versions of a project have to be known when the instance of the `ProjectData` is created. To obtain this information, the lexicographical first subcomponent of the lexicographical first component is considered. The XML files with the version names that are available there have to be representative for the whole project, thus every version has to be available. The case that by chance this subcomponent is one that has been added to the project during the development has not yet been resolved and will lead to an error.

Extracting data from Bugzilla

To extract information about the number of bugs from a Bugzilla bugtracking system, the underlying database is needed. Currently, Bugzilla can use MySQL and PostgreSQL as backends. In the current version of the tool it is only possible to extract data from a MySQL database. In a future version PostgreSQL will also be supported. To know how to extract the data, one has to take a look at the table structure in which the bugs are organized by Bugzilla.

To store the bugs, the table `bugs` exists. To determine whether a bug existed in a version, the columns `resolution` and `target_milestone` of the table can be used. The bug existed in a version, if one of the following two statements is false:

1. `target_milestone` is greater than the version and `version` is smaller than the version. If `target_milestone` is not greater than the version, it is already fixed. If `version` is not smaller than the version, the bug does not exist yet.

2. `resolution` is not marked as `WORKSFORME` or `DUPLICATE`. In this case, the entry is not relevant, because either the bug does not exist or it is already reported in another entry.

To obtain only the bugs that belong to a certain component, the column `component_id` can be used. Every component as an unique Id that is stored in the table `components`. To obtain this Id, the unique Id of the project that the component belongs to and the name of the component

is needed. The Id of the project is stored in the `products` table and can be obtained if the name of the project is known. Thus, if the name of the project and the name of the component are known, the Id of the component can be obtained. This way it is possible to obtain all bugs of a specific version of a component in an automated way, if only the version, the name of the component and the name of the project are known, using SQL queries.

Furthermore, the `bugs` table has a `severity` column. The values of this column can be used to divide the bugs into categories, according to their severity. Because it is possible for the admin of a Bugzilla to edit the possible severities, it is not possible to just divide the bugs into some fixed severity categories. However, the severities that a Bugzilla system contains are stored in the table `bug_severity` of the database. By reading the possible values from that table, the problem can be resolved.

The Databasebuilder uses JDBC to access the MySQL database of a Bugzilla system and to execute the queries. To do this, either a local copy of a dump from the Bugzilla system or direct (read-only) access to its database has to be available. The class `BugzillaReader` provides the functionality to connect to the database and to read the bug data from it. The reading function takes a list of `ProjectData` as input and extracts the bug data for all the projects and their components in an automated way as it has been described above. The results are stored as metrics with the name “BUGseverity” where severity is replaced by the actual severity as part of the component metric data. The sum of all the BUGseverity metrics is the total number of bugs, thus the metric BUG, introduced in section 2.1.2.

Reading own XML Database

Naturally, if all data from the various sources about projects has been imported and merged to one larger database that is handled by the program internally using the classes that were described in section 4.3.1, one wants to store this large database. The tree structure in which the data is organized internally suggests the usage of XML for this task. This stored XML database would be nearly worthless, if it was not possible to load it afterwards. The structure of the database is described in section 4.3.3.

Currently, it is read using the JDom API. However, because the databases can be huge – larger than 200 MB for one project – the usage of JDom is a problem. It is not a problem that is JDom specific, but a problem of the DOM in general when it comes to handling large databases. The reason for this i, that DOM handles all the data in RAM and furthermore has a large overhead. When using large files, this can become a serious problem. Because of this, the parser has to be changed to a SAX based parser in a future version.

4.3.3. The data export package

Storage as XML database

As it was mentioned above a method to store the measured data is needed. This is done using XML. It is a more or less direct mapping from the internal tree structure in which the data is organized to XML. The root of the XML document is a simple node called `database`. The children of the root node are the projects that are represented by their instance of `ProjectData`. The name of these nodes is `project`. If the project itself was the root node, it would only be possible to store one project in such a database. The project nodes have the name and the version of the project as attributes. The children of the project nodes are the measured versions and the nodes are named `milestone`. For each measured version such a node is created. The name of the version is stored as an attribute of the `milestone` node. The children of the `milestone` nodes are the nodes of which represent the `MetricDataSet` instances that are the children of the `ProjectData` instance. For each such `MetricDataSet` a node with the name of its type is created. The name of the `MetricDataSet` is stored as an attribute of that node. Its children in the tree structure are stored in the same manner as it is done with the children of the `ProjectData`. In the following part, the `project`, respectively the node that is generated by a `MetricDataSet` is called entity-node. This has nothing to do with XML entities.

The measured values are stored in `value` nodes. These nodes are the children of the entity-node to which the metric values belong. If they belong to the project, they are children of the `milestone` node of their milestone. A `value` node stores the name of the metric it represents as an attribute, the measured value is stored as the text value of the node. An simplified example with fictional metric values is given in listing 4.2. If one considers the structure of the data, it is organized by the following attributes, ordered by their priority:

1. The highest ranking attribute that organizes the data in the XML file is the project. This is represented by the project nodes being the children of the root node. The reason for this is that the measured data shall be used to analyse software projects. Thus it makes no sense to store the data in a way that is similar to what the Eclipse Metrics Plug-in does (see section 4.3.2 and listing 4.1).

2. After the projects, the data is organized by the measured version using the `milestone` nodes. It would be possible to push this down to the storage of the metric values. However, if one wants to analyse the data of a specific milestone it is easier this way.

3. The third attribute to order the data is the hierarchial tree structure. The reasons for this are the same that were given in section 4.2 for choosing an tree-like structure for the internal representation.

To write the database, JDom is used. When it comes to writing huge database, JDom has the same disadvantages as when it comes to reading. These disadvantages where already

discussed in section 4.3.2. Thus, like the reading of this database, the writing has to be changed to a SAX writer in a future version.

```
<?xml version="1.0" encoding="UTF-8"?>
<database>
  <project name="JDT" version="3.2">
    <milestone name="M0">
    <value name="TLOC">8783</value>
      <component name="Debug">
        <value name="TLOC">8783</value>
        <package name="org.eclipse.jdt.internal.launching">
          <value name="NOC">41</value>
          <class name="Standard11xVMType">
            <method name="getDefaultLibraryLocations">
              <value name="MLOC">6</value>
            </method>
          </class>
        </package>
      </component>
    </milestone>
  </project>
</database>
```

Listing 4.2: A short sample of a XML database as it is stored by the Databasebuilder. The database consists of project nodes. For each measured version a milestone node exists. For each MetricDataSet a node with the name of the type of entity of the data set is created. The name of the entity is stored as the name attribute of the node. The measured values are stored in the value nodes and belong to their parent. The name the metric are stored as attribute of the value node, the value of the metric as its text node.

Export of specified metrics to the ARFF format

With the Attribute-Relation File Format (ARFF) a format is defined that can be used as input for the machine learning tool WEKA [17]. Another advantage of this format is that it is similar to Comma Separated Values (CSV). The only important differences between ARFF and CSV are that in ARFF comments are possible and more importantly, that in ARFF a name and type for the columns can be defined. This is done using a header. Furthermore, ARFF has a fixed number of entries per line, which needs not be the case in CSV files.

ARFF is a flat format and does not support tree structures. The intended usage of this format is to extract parts of data from the whole database, to analyse them. Because it usually does not make sense to analyse data about methods and about packages at the same time, the first thing that has to be specified for the export is the type of the entity from which the metric data

should be exported. Additionally, one usually does not need all the metrics that were measured over an entity for the intended analysis. Thus, it is possible to specify the set of metrics that shall be exported.

The input of the ARFF export functionality, as it is implemented in the class `WekaWriter`, is the name of the entity type and the set of metrics that should be exported. The class is called `WekaWriter` because ARFF was designed for usage with the Weka machine learning tool. The result is a valid ARFF file where the first columns are the metric values and are named accordingly to the metric names. The last two columns contain the version and the name and type of the entity. The type of the entity is implicitly given through the type of the last column. A short example for an ARFF file can be found in listing 4.3.

```
@relation relationname
@attribute TLOC numeric
@attribute milestone string
@attribute project string
@data
582972.0,M0,JDT
583836.0,M1,JDT
```

Listing 4.3: A short sample of an ARFF file. It contains only the metric TLOC of the project JDT with the two versions M0 and M1.

4.3.4. The console and the commands

To introduce a layer that separates the program logic from the user interface a *console* and *commands* were introduced. The Java classes and interfaces that implement this concept are organized in the packages `de.ugoe.cs.swe.databasebuilder.console` and its subpackage `de.ugoe.cs.swe.databasebuilder.console.commands`.

The console is used as an observable as part of the observer pattern [9]. It is implemented in the `Console` class. When an output should take place, a message is send to the console. This message is passed on to all observers that are registered. The observers must implement the interface `ConsoleObserver`. It defines the messages that the observers must be able to handle. The observers themself are the UI classes and handle the output. To both make the console publicly available and to ensure that there is only one instance of `Console` it is implemented as a singleton. Additionally, the class is not intended to be subclassed. First of all, subclassing is a general problem with singletons and secondly, a subclass of `Console` would be confusing because it would destroy the concept of the single point of output mechanism for the logical part of the software. Currently, only one observer is implemented in the class `TextConsole`. It simply displays the send message using the `System.out.print` functionality. The versatility of this concept allows simple adding of another graphical user interface by adding a new observer.

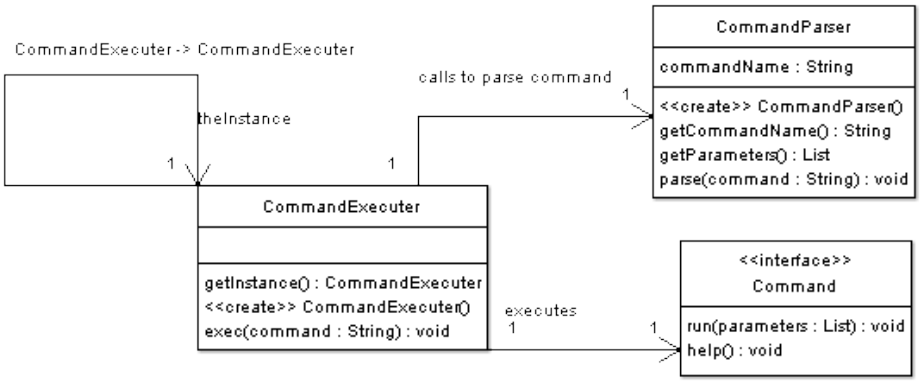

Figure 4.2.: This UML chart shows the relationship between the classes that are responsible for calling commands. The CommandExecuter uses the CommandParser to parse the command and the executes the command.

While the observer concept is used to make the output independent of the user interface and thus decouple the user interface and the program logic, the concept of commands has two purposes. The first is to do the same for the input. The second is to make it easy to add new logical functions without making big changes at the user interface. To implement both purposes, the command pattern (see [9]) and reflection are used. The pattern decouples the caller and the command, while the reflection mechanism of java is used to load the command.

For the handling of commands the class `CommandExecuter` was implemented. If the logical part of the program shall execute any kind of action - like import/export of data for example - the `CommandExecuter` has to be used. All information regarding the command, like the name of the command and possible parameters are passed as a string to the `CommandExecuter`. The form of the string has already been introduced in section 4.2. The command executer than parses the string using a `CommandParser`. Than it tries to execute the command. How this is done should be clearer, when an example is considered. The `CommandExecuter` is implemented as a singleton. Furthermore, it uses a monitor to guarantee that only one command is executed at a time. This way, only one command may edit the currently known set of problems at a time, which prevents race conditions.

The following example demonstrates how a string is parsed and used to execute a command. Let `save database.xml` the string that is passed to the `CommandExecuter`. The parser will recognize `save` as the name of the command and `database.xml` as a parameter. To execute the command, the command parser tries to load the class `CMDsave` of the package `de.ugoe.cs.swe.databasebuilder.console.commands`. If this class does not exist, the command executer will send a message to the console that the command is not known. Otherwise it tries to create an instance of the class and cast it to `Command`. If the `CMDsave` class

implements the `Command` interface, this is not a problem, otherwise an exception will cause the command to fail. If this is successful, the `run` method of `CMDsave` can be called and the command will be executed.

As the example shows, the usage of the command pattern decouples the user interface that triggers the execution of a command and the program logic that perform actions on the data. By the further usage of reflection, the creation of new commands that can be handled is simplified to a minimum. The only task is to create a new class that implements the `Command` interface and starts with `CMD` and the new command can be used. There is no need to edit the `CommandExecuter` in any way to make the command known. A list of the currently implemented commands is given in the following section.

4.4. The available commands

The implemented commands can be divided into four general categories: the commands that load data and add information to the current database of programs; the commands that export information from the database and store it as a file; the commands that make changes on the current database; the commands that display information about the current database.

---

add_bugdata_from_bugzilla

Syntax: `add_bugdata_from_bugzilla url database username password`

Categorie: Data import

This command adds metrics about the number of bugs to the current set of projects. It automatically checks for all projects if bug data is available in the database. If so, it will add it separated by severity to the current data.

`url`
: The URL of the SQL database.

`database`
: The name of the database.

`username`
: The name of the user used to access the database.

`password`
: The password of the user.

---

calc_avg
Syntax: `calc_avg type subtype metricName avgMetricName`
Categorie: Data manipulation

Calculates the average over all metric values of specific metric and stores it in one of its parents.

`type`
: The type (level) over which the average will be calculated.

`subtype`
: The subtype where the average value will be stored.

`metricName`
: The name of the metric over which the average will be calculated.

`avgMetricName`
: The name of the metric that stores the average value in the subtype.

calc_sum
Syntax: `calc_sum type subtype metricName sumMetricName`
Categorie: Data manipulation

Calculates the sum over all metric values of specific metric and stores it in one of its parents.

`type`
: The type (level) over which the sum will be calculated.

`subtype`
: The subtype where the sum will be stored.

`metricName`
: The name of the metric over which the sum will be calculated.

`avgMetricName`
: The name of the metric that stores the sum in the subtype.

calc_weighted_sum
Syntax: `calc_weighted_metric type subtype [metricNames] [weights] sumMetricName`
Categorie: Data manipulation

Calculates the weighted sum over all metric values of the specified metric and stores the result in one of its parents.

`type`
: The type (level) where the weighted sum will be stored.

`subtype`
: The subtype (level) of type, over which the weighted sum will be calculated

`metricName`
: The name of the metric over which the sum will be calculated.

`avgMetricName`
: The name of the metric that stores the sum in the subtype.

exec
Syntax: `exec filename`
Categorie: Uncategorized

Executes a script of commands as it is described in section 4.3.4.

`filename`
: The filename and path of the script.

exit
Syntax: `exit`
Categorie: Uncategorized

Exits the program.

list_metrics
Syntax: `list_metrics type`
Categorie: Uncategorized

List the names of all metrics that are available in the current data set for a type.

`type`
: The type (level) of which the available metrics will be displayed.

load
Syntax: `load filename`
Categorie: Data import

Loads a XML database that was created by this tool.
`filename`
The filename and path of the database.

load_from_eclipse_java_metrics
Syntax: `load_from_eclipse_java_metrics path`
Categorie: Data import

This command loads metric data that was exported using the Eclipse Metrics Plug-in. The files containing the metric data have to be stored in a folder structure as it is described in section 4.3.2.
`path`
The path of the metric data.

new
Syntax: `new`
Categorie: Uncategorized

Deletes the current set of data and initializes a new, empty one.

normalize
Syntax: `normalize type metricName newMetricName`
Categorie: Data manipulation

Calculates a new metric as the normalized values of another. More about normalization can be found in section 5.2.
`type`
The type (level) of the metric, that will be normalized.
`metricName`
The name of the metric which will be normalized.
`newMetricName`
The name of the normalized metric.

save
Syntax: `save filename`
Categorie: Data export

Saves the current set of projects and their metric data as an XML file.
`filename`
The filename and path, where the data shall be stored.

save_as_arff
Syntax: `save_as_arff filename [metricNames] type`
Categorie: Data export

Saves the values of the defined metrics as an ARFF file as described in section 4.3.3.
`[metricNames]`
The names of the metrics that shall be exported.
`type`
The level of the metrics, i.e. method, class or project.

zeroize
Syntax: `zeroize type metricName newMetricName`
Categorie: Data manipulation

Calculates a new metric as the zeroized values of another. Zeroizing calculates $v_i^{\text{new}} = v_i - \min\{v_i \text{metric values}\}$ for all values v_i of the metric, thus setting the smallest value of the metric to 0, while the absolute distances between the values are kept.
`type`
The type (level) of the metric that will be zeroized.
`metricName`
The name of the metric which will be zeroized.
`newMetricName`
The name of the zeroized metric.

5. Detecting feature freezes using the k-means algorithm

Up till now, only concepts were introduced. In this chapter everything that was introduced up to now will be used in an approach to detect a feature freeze in a project. To do this, metric data measured over the progress of a project will be used as input for the k-means algorithm.

5.1. Collecting Metric Data

The collected metric data was measured from the Eclipse Platform Project 3.2 and the Eclipse Java Development Tools 3.2. The projects are both part of the Eclipse Project hosted by the Eclipse Foundation [2]. Of both projects 14 versions were measured. What versions were measured was determined using the project plans of both projects. The project plans were almost identical, except for a few days that some versions may deviate from another. In both project plans there were 6 milestones M1 to M6 and 6 release candidates RC1 to RC6. Because both projects are not newly developed from scratch but based on their previous version 3.1, this version was measured as a basis, too. The last measured version is the release of the project.

The source code of the projects is maintained in the repository of the Eclipse Foundation. It was retrieved and measured using the concepts introduced in chapter 3. Both projects consist of several components. In these components several Eclipse plug-ins are implemented as subcomponents. Together these Eclipse plug-ins implement the larger components of Eclipse.

This causes two problems. The first problem is that there are many subcomponents and all have to be measured for each version. This results in a large amount of work. This is not a serious problem, it only increases the workload. The second problem causes more trouble. Between these subcomponents there are many dependencies. Thus, to compile a subcomponent of a version correctly, one usually needs to provide the same version of other components. This can result in a snowball effect. As a result, to measure one subcomponent, not only the source code of that subcomponent has to be retrieved, but also of others. It is not always obvious which other components are needed. The person that performs the measurement has to solve this problem for each subcomponent independently. The measurement resulted in about 1 GB of XML exported using the Eclipse Metrics Plug-in that contained metric data about the two projects.

To gather bug data, a dump of the Bugzilla bugtracking system that is used by the Eclipse

Foundation was used. The nearly 3 GB large dump is publicly available on the foundation's website. To extract the data from the database, the tool that was introduced in the previous chapter was used.

5.2. Normalization of the Data

Before the metric data can be used, it has to be prepared. To be exact, it has to be normalized. If projects shall be compared, their absolute values are usually not comparable because of their different sizes. The data will be used to analyse the progress of the projects. For the progress, the size does not matter. By normalizing all metric values, the projects become comparable. Normalization means that all metric scales are changed to the interval $[0, 1]$, while keeping the relative distances.

Another reason why the data has to be normalized is the way in which the k-means algorithm works. The cluster assignment is done using the euclidean distance from the centers that represent a cluster. Consider two metrics m_1 and m_2 , where m_1 outputs values in the interval $[0, 1]$ and m_2 in the interval $[0, 100]$. For two entities x and y these two metrics describe a feature vector $m(x) = (m_1(x), m_2(x))^t$ respectively $m(y) = (m_1(y), m_2(y))^t$. The euclidean distance between these two vectors is

$$dist(m(x), m(y)) = \sqrt{(m_1(x) - m_1(y))^2 + (m_2(x) - m_2(y))^2}. \tag{5.1}$$

Because the values of m_2 can have a greater distance between themselves, usually this distance will be dominated by the second term. Thus, the values of m_1 would be less important for the result of the k-means algorithm than those of m_2. So the importance of a metric would be defined by how large its scale is. With normalized data all metrics have the same scale and thus the same importance.

5.3. Detailed description of a feature freeze

As it was already said in chapter 2, software projects have different phases. The points of transition between these phases are often of special importance and thus have a name of their own. A *feature freeze* is such a point during software project. In every successful software project a feature freeze occurs. Either it is planned or it happens implicitly. The latest point of a feature freeze is the end of the project. How and when a feature freeze happens depends strongly on the organization of the project. Especially with agile software development the complete feature freeze will be in a very late phase of the project.

As the name suggests, the current set of features that the software provides is frozen, thus not changed anymore. This changes the focus of a project. The project enters a phase were it is stabilized. Before the stabilization, many resources are used to add new features to the

software under development, thus changing it constantly. With the feature freeze, this stops. Instead the already existing features are enhanced. This enhancement can be testing, bug fixes, refactorings or documentation. Generally, after a feature freeze occurred, there should not be many changes to the software and its source code anymore. Instead the number of bugs should decrease and the quality of the source code should increase.

5.4. Selecting the Software Metrics

In chapters 2 and 3 many software metrics were introduced. However, each metric has different characteristics and can be used to describe different features. In this case, the task is to detect a feature freeze. As it was said in the previous section, after a feature freeze, the source code should not change that much anymore. A way to measure this is to consider the lines of code. If new features are added, the lines of code increase greatly. When only bugs are fixed and the project is stabilized, the lines of code should be relatively stable.

The other important change is that the focus changes to bug fixes. So after a feature freeze, the number of bugs should decrease. This can be measured using the metric BUG that was introduced in section 3.3. The combination of these metrics can be used to measure two of the above mentioned properties of a feature freeze. This is, of course, only one possible combination of metrics. For example, if after a feature freeze during the phase of stabilization of the project it should be refactored to improve the quality of the source code, other combinations of metrics could be more effective. However, by using LOC and BUG, two features are measured that should take place after every feature freeze. Thus, for a general approach it makes sense to use only these two metrics.

5.5. Application of the k-means algorithm

The approach used in this thesis to detect whether a feature freeze actually took place or not is to apply the k-means algorithm to the mined metric data. As the k-means algorithm is an unsupervised learning algorithm, one might wonder why it is used to analyze supervised metric data. The knowledge when a feature freeze took place which is included in the project plan is intentionally “forgotten”. If the unsupervised k-means algorithm still detects it only using the metric data, this can be used as an indicator that the feature freeze actually took place. Only because the project manager said that a feature freeze should take place at some point of time, it is not said that the developers actually abided by it. In other words: even if the project plans said that a feature freeze took place at a specific version of a project it is not necessarily true. Thus, the observation of the supervised data would be wrong.

To apply the k-means algorithm, the number k of clusters that the algorithm should compute has to be determined by the user. In this work two clusters are needed, thus $k = 2$. One cluster

for the versions before the feature freeze and one cluster for the versions after the feature freeze. If a feature freeze took place, the clusters should divide the versions of the project in two sequences of versions. For example, consider a project with 20 versions. Consider three possible results:

1. One cluster contains the versions 1 to 14 and the other the versions 15 to 20.
2. One cluster contains the versions 1 to 10 and the other the versions 11 to 20.
3. One cluster contains the even numbered versions, the other the odd numbered versions.

The first two results are reasonable, but would lead to different interpretations when the feature freeze took place. The third result seems arbitrary. A result where adjacent versions alternate between the clusters is not reasonable. Adjacent versions should be similar to some degree and thus belong to the same cluster. If this is not the case, this indicates management problems. In a healthy project, progress is made steadily. If the progress is made steadily, the metric values should also be steady, not necessarily monotonic, but there should not be large jumps back and forth. If there are $k = 2$ clusters, the characteristic of the k-means algorithm is to decide to which cluster a vector belongs by its euclidean distance from the cluster centers. This makes sure that, when the development is steady, the clusters contain adjacent versions.

5.6. Results

When applying the k-means algorithm to the normalized metric data that was mined from the two Eclipse projects, the software versions are clustered correctly. The adjacent versions from the initial version to Milestone 4 are assigned to one cluster, the remaining adjacent versions from Milestone 5 to the final release to the other cluster. Thus, the result clearly separates the first versions from the later versions and detects the feature freeze that took place at milestone 5. The error of the result is 0.

	JDT Project		Platform Project	
Milestone	Lines of Code	Number of Bug Reports	Lines of Code	Number of Bug Reports
Start	582972	383	647322	767
Milestone 1	583836	383	626734	764
Milestone 2	593967	374	633419	762
Milestone 3	605037	362	646458	762
Milestone 4	620556	359	656836	757
Milestone 5	645890	355	690544	740
Milestone 6	658664	344	706153	730
Release Candidate 1	661111	340	711387	722
Release Candidate 2	667185	339	714266	716
Release Candidate 3	667592	338	714040	716
Release Candidate 4	667801	337	715910	716
Release Candidate 5	667781	337	716032	716
Release Candidate 6	667794	337	716034	715
Release	670645	337	716150	715

Table 5.1.: Metric values collected from the Eclipse JDT and Platform project. The horizontal line between Milestones 4 and 5 indicates the detected feature freeze.

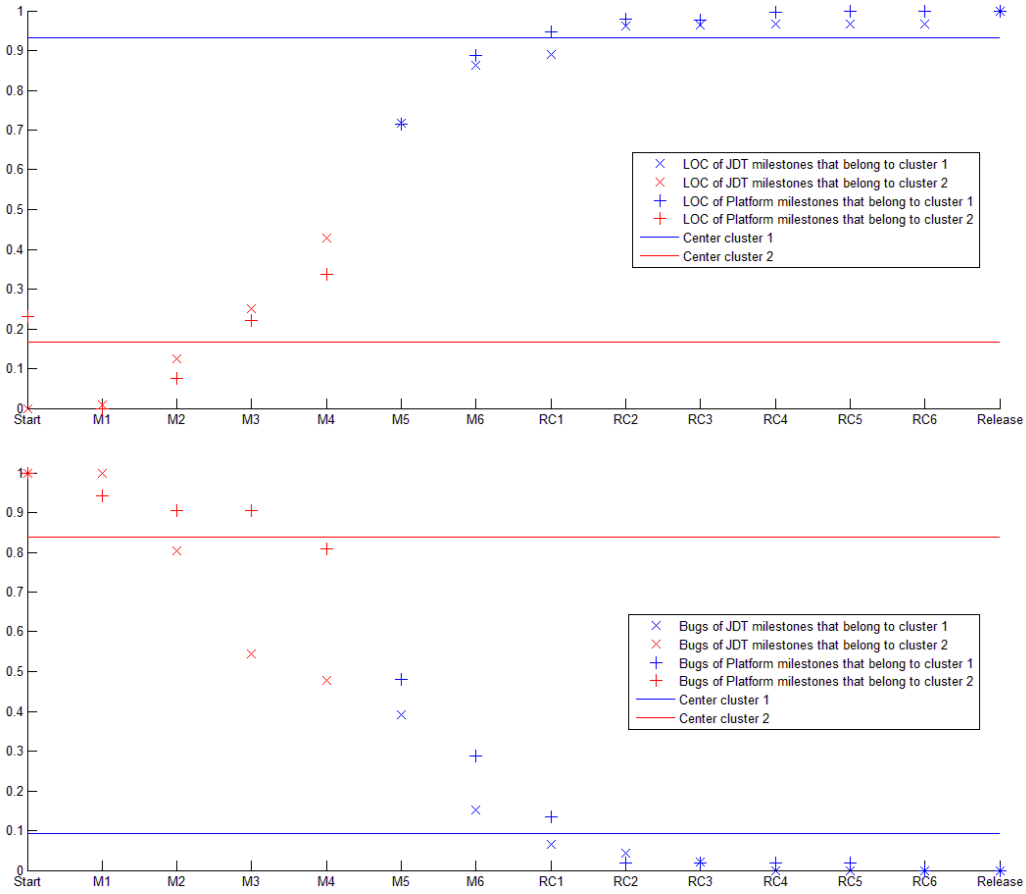

Figure 5.1.: Normalized lines of code (upper chart) and normalized number of open bug reports (lower chart) of the Eclipse JDT (marked as “x”) and the Eclipse Platform (marked as “+”) project and their development during the project. The color indicates the cluster of which a milestone is part, the two horizontal in each chart lines indicate the center of the two clusters.

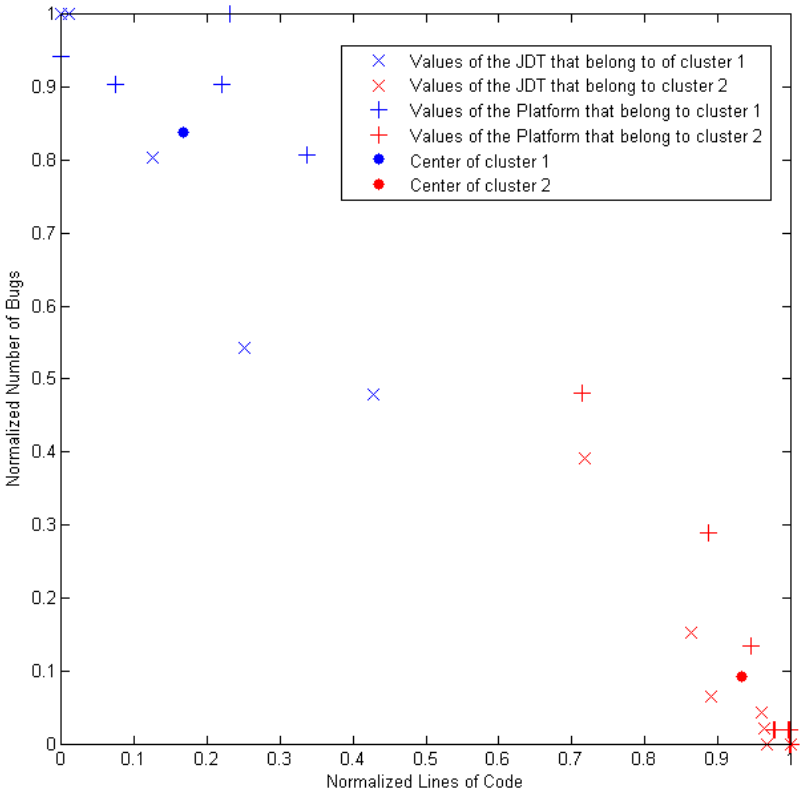

Figure 5.2.: This scatterplot shows the distribution of the metric values of the Eclipse JDT (marked as “x”) and the Eclipse Platform (marked as “+”) at the milestones. The color indicates the cluster of which a milestone is part, the dots mark the center of the two clusters.

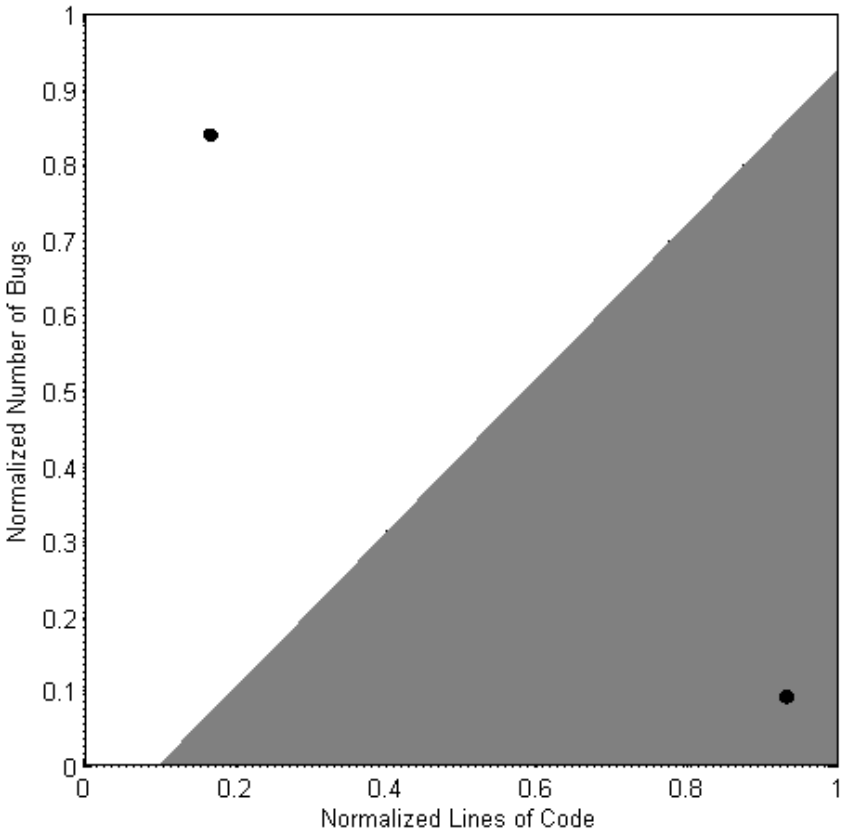

Figure 5.3.: This figure shows a Voronoi diagram of the resulting clusters. The white area marks all pairs of metric values that would be considered as before the feature freeze; The gray area marks all pairs of metric values that would be considered as after the feature freeze. The black dots mark the centers of the clusters.

6. Conclusion and Outlook

As the successful detection of the feature freeze shows, it is possible to detect crucial events during the progress of a project by applying the k-means clustering algorithm to metrics. This allows to determine in retrospective whether a feature freeze failed or was successful. The experience gained from this insight can be used to improve the development process for future software projects. If a feature freeze failed, the mistakes that lead to this failure in the project management can be detected and eliminated, improving the chance of success of other projects.

To apply this approach, metric data measured during the development of a project is needed. To gather this metric data in retrospective is a lot more work than to gather it as part of the quality management of the project. If the metric data is available, it is simple to apply the k-means algorithm to it. Maybe some work needs to be done to normalize the data, but that is a simple task compared to the measurement itself. To gather metric data during a project to observe it and use the data to improve the development process is nothing new. For a company to obtain CMMI [16] level 4, the usage of a consistent set of metrics to observe the projects is a requirement. To obtain level 5, this data has to be constantly used to improve the quality of the development process. Thus, a quantitively managed development process already gathers the required metric data. The k-means algorithm can be used as a simple, but effective approach to assess one attribute of the development process.

There are some problems with this approach. The currently largest problem is the lack of data. The experiments were performed only for two projects from a similar environment. This lack of data presents a problem for the plausibility of the results. This can easily be resolved by collecting more data. The only problem is that it is very time-consuming to do. However, as more data is collected over time, this problem will resolve itself. The other big problem is that the k-means algorithm always divides the data into k clusters. Thus, in the approach discussed in the previous chapter, the measured versions will always be divided into two clusters. This does not depend on whether there actually was a feature freeze or not. This leads to a very problematic question: Is there a meaning in the separation of the data at a certain point or is it random?

This question cannot be answered in a simple way. The separation of the data as it is done by the k-means algorithm is always an indicator that at the point of separation something took place. If this coincides with a crucial point during the project, it is an indicator that this crucial event actually had an effect and was not only supposed to take place according to the project plan. Thus, the significance of the result also depends on the person who interprets it. To

improve the quality of this indicator, the k-means algorithm can be applied repeatedly with random start centers. If it outputs the same result regardless of the start centers and the separation is stable, it improves the indicator's quality. If the result is not stable, there is no clear separation in the data, but it depends on the random factor of the start centers. Thus, there was no clear change in the metric values, which is an indicator that no feature freeze took place.

The next problem that has to be addressed is that it is not clear *what* the algorithm detects. It depends on the data and the user interpretation. The metrics have to be chosen in a way that they can reflect the feature the user is looking for. For the case study performed in this work the metrics LOC and BUG were selected because they reflect two features that are significant of a feature freeze. Even so, the results can be misleading. For example, if a huge component is integrated in the software, the lines of code will be significantly higher after the point of integration. The same would be the case with the number of bugs, if the components were buggy. As the k-means algorithm uses distances, it is probable it will separate the versions at the point of integration. The same could be the case if a component was replaced or completely removed. In this scenario the algorithm would detect a crucial point in the project. However, there is no need at all to confirm that it took place. The integration of a component is hardly something that can be missed, so it does not need to be confirmed. The person who interprets the result of the k-means algorithm has to be aware of such developments during the project and needs to consider them when interpreting the result.

Because of these circumstances it is not possible to simply write a guide how the result can be interpreted and what it means. The reason is the uniqueness of every software project. Thus, the interpretation is also unique to the project. The general result is what is important. Using only metric data measured at different points of time during a project, it is possible to detect features of the project with learning algorithms. It is even possible in an unsupervised setting. So even if this approach has certain problems, the result is an indicator that more research in this area will lead to good results. In the following, an outlook on some possible learning scenarios is given.

The obvious thing to do is to use other clustering algorithms. There are several scenarios in which other clustering algorithms can be used. The first is to test whether the result is stable or not. Applying different clustering algorithms that detect two clusters can replace the repeated execution of the k-means algorithm to test whether the result is stable or not. If all algorithms compute the same result, it is stable not only for the k-means algorithm but for a whole class of algorithms. This would improve the quality of the result greatly.

It would be interesting to vary the number of clusters, too. For example a third cluster could be used for a transition phase. In this case, the first cluster would contain the phase where many new features are developed, the second cluster for the phase when the features are stable and the project is stabilized and one cluster for the transition between these two phases, where most features are already stable and only some still under development. This third cluster is an acknowledgement that most projects have problems with keeping their deadlines and thus the feature freeze is often not realized for all features at the same point in time.

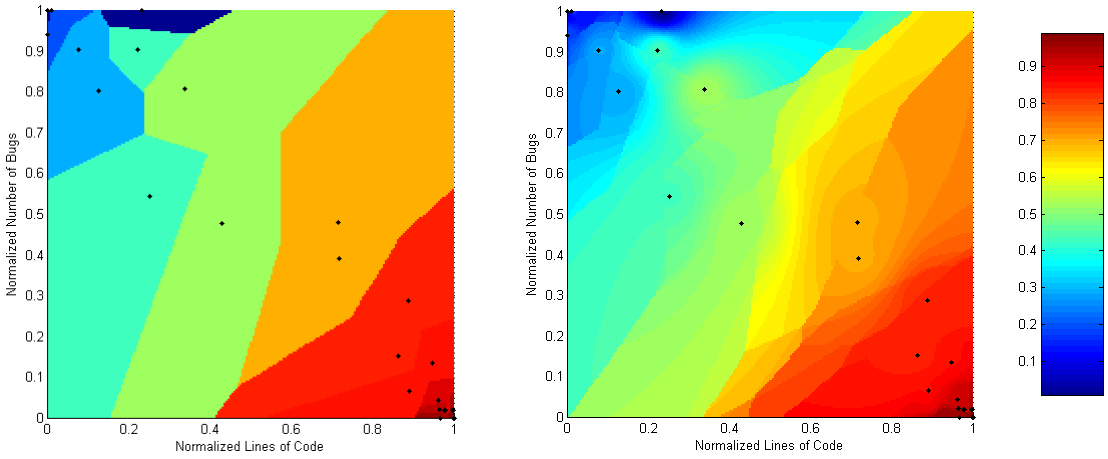

Figure 6.1.: These figures show how the nearest neighbour algorithm can be used to predict the project progress using metric values. As input the normalized values of the metrics LOC and BUG are used. The color indicates the progress of the project: blue is the start of the project, red is the end of the project. The black dots indicate the position of the known values, that means the values measured from the Eclipse JDT and the Eclipse Platform Project. On the left: the simple nearest neighbour algorithm; On the right: the distance weighted k-nearest neighbour algorithm with $k = 3$.

The two other applications of cluster algorithms above still suffer from the issue that they use a fixed number of clusters. Thus, the problem that the algorithms will always detect something remains. A solution to this problem is the third variation of applying cluster algorithms: using an algorithm that detects the number of clusters itself, like the EM algorithm. Such an algorithm does not have the problem that it *has* to find clusters.

The above approach can only be used in retrospective. To check a project that is still under development, it cannot be applied. A different direction of research would be to use supervised learning algorithms to calculate such a predictive model. This way it may be possible to calculate a hypothesis that can be used to discriminate versions using the metric data directly into versions that occurred before a feature freeze and versions that occurred after a feature freeze. Possible approaches would be the calculation of a hyperplane. The two half spaces defined by the hyperplane would be the time before, respectively after the feature freeze. Another possibility would be to use a Support Vector Machine (SVM). If the time before and after a feature freeze is interpreted as a *state* that the project is in, Conditional Random Fields (CRF) [13] can be used to predict the state of a project.

Another supervised approach would be to use a function learning algorithm. This way a model on how a project should progress in the process model could be extracted from measured values of old projects. This model can then be used to predict the progress of a project. Figure 6 shows how the nearest neighbour algorithm can be used to calculate a model how projects that are developed using the same process as the two measured Eclipse projects should progress.

Actually, in practical approaches to determine the status of a project, managers use the nearest neighbour algorithm intuitively. To assess the status of a project they use their experience from other projects that were in a similar stage. This is exactly what the nearest neighbour algorithm does.

These are only a few possible approaches that could be applied to this kind of data. With more data, more approaches become possible, for example if data about test coverage or test results is also available. This shows how much potential is in this kind of research. The most important message of this work is, that even the simple approach of using only two software metrics and the relatively simple k-means algorithm are able to detect a feature of something as complex as a project is. The future will show what other approaches are capable of.

A. Table of Acronyms

ARFF Attribute-Relation File Format

BUG Number of Bugs

CRF Conditional Random Fields

CSV Comma Separated Values

DOM Document Object Model

EBNF Extended Backur-Naur Form

LOC Lines of Code

RAM Random Access Memory

SAX Simple API for XML

SVM Support Vector Machine

XML Extensible Markup Language

List of Figures

List of Tables

Bibliography

[1] Bugzilla. http://www.bugzilla.org.

[2] Eclipse Foundation. http://www.eclipse.org.

[3] Eclipse Metrics Plug-in. http://metrics.sourceforge.net.

[4] JDom. http://www.jdom.org.

[5] PROMISE. http://www.promisedata.org.

[6] Ieee standard glossary of software engineering terminology. Technical report, 1990.

[7] K. Beck. *Extreme programming explained: embrace change*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[8] N. Fenton and S.L. Pfleeger. *Software metrics: a rigorous and practical approach*. PWS Publishing Co. Boston, MA, USA, 1997.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Reading, MA, 1995.

[10] B. Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1995.

[11] B. Hindel, K. Hörmann, M. Müller, and J. Schmied. Basiswissen Software-Projektmanagement, 2004.

[12] IEEE. Ieee glossary of software engineering terminology, ieee standard 610.12. Technical report, IEEE, 1990.

[13] J. Lafferty, A. McCallum, and F. Pereira. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. *MACHINE LEARNING-INTERNATIONAL WORKSHOP THEN CONFERENCE-*, pages 282–289, 2001.

[14] T.M. Mitchel. Machine Learning. *MacGraw Hill*, 1997.

[15] I. Sommerville. *Software Engineering*. Addison-Wesley, 2006.

[16] C.P. Team. Capability Maturity Model® Integration (CMMI SM), Version 1.1. *Pitsburg, Software Engineering Institute*, 2001.

[17] Witten, I and Frank, E. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition*. Morgan Kaufmann, 2005.