MASTER'S THESIS 2020

Automated Quality Assessment of Firmware Releases

Dragan Adzaip, Sven Andersen

ISSN 1650-2884
LU-CS-EX: 2020-30

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-30

Automated Quality Assessment of Firmware Releases

Dragan Adzaip, Sven Andersen

Automated Quality Assessment of Firmware Releases

Dragan Adzaip
dat14dad@student.lu.se

Sven Andersen
jap13san@student.lu.se

June 23, 2020

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Martin Höst, martin.host@cs.lth.se

Examiner: Elizabeth Bjarnason, Elizabeth.Bjarnason@cs.lth.se

Abstract

Some companies strive for the best quality in their releases. For some companies the firmware quality can be "good enough" to be released. And others just want to get new functionality out quick. No matter the case, with the high tempo in the industry, with frequent releases and constantly having to be up to date, it is hard to stop and evaluate the quality. If customers or users give feedback, the companies have some idea of how the product is doing. However, this can only be seen after the release, given that there is any feedback. Is it possible to gather data from previous releases and predict the quality of the next release before it is released? Is it possible to determine or identify issues earlier to avoid problems when the customer is using the product? This thesis investigates how to create a model that is able to predict the quality of a release. The help from this kind of assessment would allow the company to measure how their firmware is doing before a release. The end product of the work are three models that are able to indicate the quality of a firmware release. Based on the data available before release, it is able to give one or several metrics that can be used to predict how well the release would fare if shipped out. We made a program for applying this model, so that it can be used at Axis. The way we did it was by first doing a literature study as well as an in-house research at the company, to find relevant metrics and data to use. We then proceeded to extract the data we decided to continue with. That data was based on said literature study and primarily on discussions and interviews with relevant personnel. After that, by using linear regression, we created several models where each model used different metrics. The models are in fact just functions where the metrics are unknown variables and their values in-parameters. Based on the values of the metrics, the model will predict a value corresponding to the quality. Lastly, the models were evaluated by training them on all but one data point and then predicting on that missing point. The three best were presented to the case company. They were satisfied in the end. They realised that there was too little data to make a super model of this kind, but will continue to use and further develop them to improve their accuracy.

Keywords: quality assessment, firmware, release, metrics, models, prediction

Acknowledgements

We would like to thank our supervisor at LTH, Martin Höst. Martin has helped us with all things regarding the writing of the report as well as always providing tips and guidance. We would also like to especially thank our supervisor at Axis Communications, Sridevi Uppuluri and Jesper Haine, for always helping us and answering our questions. We would also like to send our thanks to everyone else that we have been in contact with at Axis, for their very friendly reception and their eagerness to help. We have had a great time working with you all, and we hope to see you again. Thanks again to Axis, for all the free coffee, soda, and bananas that supplied us with energy during our work.
We want to thank Javahuset, also known as Cykelhuset, also known as Tidshuset. The greatest group there ever was. You know who you are. Finally, we want to thank our families for being the best moral support, not only through this thesis but also through the whole time at LTH.

Contents

Chapter 1
Introduction and Aim

he Currently many companies frequently make new releases and ship them to customers. When that is done the development process seems finished. However, the question "how good was the release?" is rarely asked, unless users give feedback which can be both good or bad. Could information about the release help in decision making to improve the quality of future releases or direct the work in a specific path? By gathering data from earlier releases and internal products, is it possible to predict how the next one will be or determine and identify issues earlier to avoid problems when the release is already in the customers hands? Visualising data trends for example, has been shown to have a large impact in supporting the daily work, branch merging and at release time in a case study done at *Westermo Network Technologies AB* [31].

1.1 Case company

The case company sell cameras but do not build them in their main office. Instead they write the code which controls the cameras, called firmware there. There are several problems that arise when releasing bad firmware. A major one being loosing time and resources to unplanned and unexpected obstacles. If the firmware is bad then actions has to be taken to improve it, which means resources has to be spent, resources that could be invested in more important tasks. Resources could be manpower, or material amongst other things. Instead of working on new projects and advancing the department or company, they have to slow down and redo tasks that should already be finished. This can result in shorter deadlines for the next release. It can result in customers being unsatisfied and a negative reputation amongst other potential customers. Manpower and material also do not come free, which could be avoided or at least be deployed on different enterprises. For companies that seldom make releases and tend to make them larger this problem potentially becomes a bigger issue than for companies that make smaller but more frequent releases. This is because of two reasons. Firstly, if a release is big it could potentially contain more flaws which requires more

resources to fix. Secondly, if the company waits long before the next release and the flaws are major, customers will be unsatisfied with the waiting time before getting their new update.

1.2 Quality definitions

According to Nagappan et al. [20] estimates of software field quality in the industry are often available too late to affordably guide corrective actions to the quality of the software. True field quality cannot be measured before a product has been completed and delivered to an internal or external customer. Here, the quality is measured using the number of failures found by the customers and since this information is only available late in the process, corrective actions tend to be expensive [20]. What Nagappan et al. says applies for Axis as well. Developers, but also other personnel can benefit from an early warning regarding the quality of their product.

The ISO/IEC 9126 defines quality of software products as *"the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs"*. Furthermore, the ISO/IEC 9126-1 specifies two distinct models for software quality. Internal and external quality is modeled with the set of the following characteristics: functionality, reliability, usability, efficiency, maintainability and portability. Whereas quality in use characteristics are modeled with four other characteristics: effectiveness, productivity, security and satisfaction [22].
The standard also says that evaluators, testers and developers will be able to evaluate external software product quality and address quality issues. For higher positioned personnel, it would be of interest in evaluating "quality in use" since they inform on the decision making process. Quality in use is the combined effects of the six categories of software quality when the product is used. The ISO/IEC 9126 defines it as the capability of the software product to enable specified users to effectively, safety, satisfactory and with productivity, achieve specified goals in a specified context of use [22]

1.3 Model

The main part of this thesis is the model which is intended to help Axis in decision making. It takes a number of inputs(Q_{in}) and produces an output depicting quality(Q_{out}), as shown in Figure 1.1. The model is in fact just a mathematical function where the metric's values are in-parameters. We found out there are many ways to design the model, each way explained more in detail by Li [13]. The one we are interested in however, is using *linear regression*. These modeling methods are ways to produce models using historical information on predictors such that the resulting model can produce a prediction given the predictors values for a new observation.

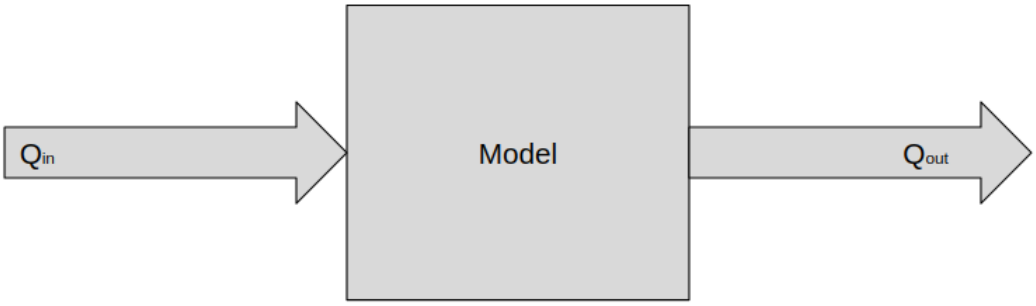

Figure 1.1: Model

For the model described in Figure 1.1 we need an input which we call Q_{in}, also known as predictors or independent variables in the literature. The input can be one or several metrics, that are deemed significant enough to be input. For instance if there is a metric that never changes then it will not matter for the model since it is always going to contribute with the same result. The second part of the model is its output which we call Q_{out}. This is known as dependent variables in the literature. The output is mainly decided by what Axis wants since it is supposed to give a prediction on their releases. There are different ways of presenting the output and it is also dependent on the input to the model. For instance if the probability of something is desired as output, then the model should work with numbers as input to be able to calculate a probability. If only a pass or fail is desired then boolean values might be enough as input.

We made a program for applying this model, so that it can be used at Axis. The main method of the thesis was a case study on the company to be able to create the model, more detailed in Section 4.1. We started by doing a literature study as well as an in-house research at the company, to find relevant metrics and data to use. We then proceeded to extract the data we decided to continue with. That data was based on said literature study and primarily on discussions and interviews with relevant personnel. By using linear regression, we created several models where each model used different metrics. Lastly, the models were evaluated by training them on all but one data point and then predicting on that missing data point. The three best models were presented to the case company.

1.4 Report structure

The first three chapters provide introduction, a glossary and background for the thesis. In Chapter 4 our method and decisions are presented. Chapter 5 shows our results and Chapter 6 and 7 summarises the thesis with discussion, conclusion and future work.

1.5 Distribution of work

The work has been evenly distributed between the authors. Both have written on the report continuously and done some programming. Both have also been present at all meetings and

other discussion sessions needed to move the research forward. Sven Andersen has focused a little bit more on programming and calculations. Dragan Adzaip has focused on report writing and keeping things structured to fit according to Axis standards, such as code standards, database queries and data formats, since he had some experience from the company earlier. Towards the end we worked more and more together to streamline the work. We pair-programmed, wrote the report and other documentations together and made sure the handover of the code went smoothly.

Chapter 2
Glossary

This glossary contains words and terms that are used in this report.

General terms:

- Feature: A feature is a functionality or ability of a camera that is included in a release.
- Main/Master branch: The branch that holds the latest firmware for the products. Since the case company wants to make frequent releases the goal is to always have a stable and clean master branch so that the release branches do not require too much effort to fix if necessary.
- Metrics: A characteristic which is measurable or countable. Used to, for instance measuring software performance, planning work items or measuring productivity.
- White noise: White noise is a term mostly used in signal processing, and is similar in nature to grey noise. This term is used in our text to signify background interference and inaccuracies in sample values.

Code terms:

- Assertion: A test condition in a test case, that is making sure that the software is functioning as expected.
- CBO: Coupling between object classes. A count of the number of classes that are coupled to a particular class, i.e. where the methods of one class call the methods or access the variables of the other.
- DIT: Depth of inheritance tree. The maximum length of a path from a class to a root class in the inheritance structure of a system. Measures how many super-classes can affect a class.

- kLOC: Thousands of lines of code in the project.
- SLOC: Source lines of code. Lines of code in the source code.
- TLOC: Test lines of code. Lines of code regarding tests.
- WMC: Weighted methods per class. Measures the complexity of a class. For instance by the cyclomatic complexities of the methods in the class. A high value indicates a more complex class.

Axis terms:

- CST (Customer support ticket): There are two systems where the term CST is used. The first is the customer support interface, where customers can get support with problems, in this system the tickets are written by the customer. The second system is the internal ticket system, where a ticket is composed by an employee. These tickets can be based on tickets from the first system, and are then also CST tickets, as they are indirectly written by a customer.
- Confluence: A software tool similar to Wikipedia where teams create, collaborate and organize all their work in one place.
- External data: External data is similar to internal data, except that the cameras are actually those that are run by customers. Customers have the option to automatically send data from their cameras, this is the data which is known as external data.
- IDD (In-Device Diagnostics): Is a department and framework making it possible to monitor embedded products. They monitors installed products and gather usage data to:
  - Know what is important for the customers.
  - Tighten the development feedback loop.
  - Detect issues before the customers do.
  - Simplify troubleshooting.
- Internal data: Internal data is the data that is collected from cameras that are within the grounds of the case company. This data is slightly different from test data, since there are no specific areas of function that are being tested. It is a way to run the latest software under somewhat realistic conditions, without subjecting the customers to possible faults.
- LTS (long term support): Are special releases that will be kept active for a long time. This is done for several reasons, but one is the limitations of older hardware. The older hardware might not be able to accommodate the newer features or firmware sizes, so as a solution, every once in a while a release is chosen as LTS. The LTS will only be upgraded with essential updates, such as security patches.
- Trouble API: Is an API used to communicate with the ticket system Trouble. From the API it is possible to get various ticket information such as categories, departments, packages, products and users amongst others, as well as whole tickets.

Our terms:

- Package changes: Refers to a the software packages that are the components of the software that is being run on the cameras. By changes, we refer to both changes in the code and version number.
- Pulled products: When a release is planned, it consists of several parts. One of these parts is a list of products that are intended to be included. Later down the release process, some of the products that were originally supposed to be part of the release, will for some reason be excluded from the release. Or, in terms of set theory, it is the symmetric difference of the planned release products and the actually released products.
- final to final: Means the interval in time between one final release and the next final release. The final comes last, i.e. after eventual alphas, betas, etc.

Chapter 3
Background

What is truly a good release? The definition is ambiguous and may vary from company to company. It may differ even more if the companies work with different sorts of end product. This thesis is written for Axis Communications, which sells cameras, The performance of the cameras is essential and it also becomes one way of measuring quality. For example, if the memory usage and boottime is low for a camera, then one can perhaps say that it is good. On the other hand, companies that do not work with cameras do not have the possibility to measure quality in this way, but perhaps in some other way that Axis can not. In this thesis we have two aspects to how the theory applies. Things that only work for Axis, such as specific metrics mentioned above, and general things that can be seen or used in all software companies such as lines of code for example.

This chapter will look into the background, starting at a general level with previous similar work. After that follows a section about linear regression which is used to create the models. Two shorts sections about AHP and black box testing are included, where AHP is a technique we used during the case study and black box testing is a similar strategy to our model. The chapter ends with a section that brings up things that are more specific to our case.

3.1 Quality models

The subject of quality prediction is not very prevalent in research, thus finding related work proved a challenge. Furthermore, no company is identical and quality definitions differ. Even so, related work does exist. Nagappan et al. [20] did a very similar case study to this one where they estimated software quality using in-process testing metrics . Earlier they had created a metric suite called "Software Testing and Reliability Early Warning metric suite for Java" (STREW-J) [19], which is applicable for development teams that write extensive automated test cases. STREW-J leverages the utility of automated tests suites by providing a post-release

field quality estimate. The field quality estimate relative to historical data is calculated using multiple linear regression analysis which is used to model the relationship between software quality and selected software metrics. The metric suite consist of nine metric ratios which are shown in Table 3.1.

1	nbr assertions / SLOC
2	nbr test cases / SLOC
3	nbr assertions / nbr test cases
4	(TLOC/SLOC) / (nbr classes$_{test}$/$nbrclasses_{source}$)
5	$\sum cyclomaticcomplexity_{test}/\sum cyclomaticcomplexity_{source}$
6	$\sum CBO_{test}/\sum CBO_{source}$
7	$\sum DIT_{test}/DIT_{source}$
8	$\sum WMC_{test}/\sum WMC_{source}$
9	SLOC / min SLOC

Table 3.1: Table showing the nine metric ratios used in the metric suite by Nagappan et al. [20].

Each metric makes an individual contribution towards estimation of the post-release field quality, but works best when used together. Development teams record the values of these metrics and the actual post-release field quality of projects. The historical values from prior projects are then used to build a regression model that is used to estimate the post-release field quality of the current project under development. The metrics are a bit hard to compare with the ISO 9126 standard metrics described in Section 1 since the metric ratios are on a lower level, i.e. more close to the code. However, some of them have a relation. For instance, ratio 5, 6 and 8 talk about complexity. A parallel can be drawn with maintainability and portability with the argument that the more complex the code is, the harder it is to maintain and port to other systems. Furthermore, ratio 2 and 3 talk about tests. More tests does not necessarily mean higher reliability and better functionality. It depends on the tests. But if the tests are written right, it should contribute to reliability and functionality. The last metric is a relative size adjustment factor. Size can always be discussed and defect density has been shown to increase with class size [5]. There are also different kind of sizes, for example lines of code, class size and release size amongst others. Nagappan et al.[20] account for the difference in size in terms of lines of code for the projects used to build the STREW prediction equation using the size adjustment factor .

When deciding what metrics to choose, they consulted Peng and Wallace [24] who listed 5 important characteristics for a good metric. These points were written in 1993 but are still relevant to our study in the current year.

- Simple - definition and use of the metric is simple.
- Objective - different people will give identical values. This allows for consistency, and prevents individual bias.
- Easily collected - the cost and effort to obtain the measure is reasonable.
- Robust - metric is insensitive to irrelevant changes, allows for useful comparison.

- Valid - metric measures what it is supposed to, this promotes trustworthiness of the measure.

The study was carried out in a junior/senior level software engineering course at a university in the United States. The students developed an open source plug-in in Java that automated the collection of static code metrics. A total of 22 projects were submitted and used in the analysis. The size of the projects were measured in lines of code with the largest project having 3631 SLOC and 2115 TLOC. The mean of all projects was 1996.9 SLOC and 688.7 TLOC respectively. The projects were evaluated using a set of 45 black-box [21] test cases. The post-release field quality was calculated via the black-box test cases as the ratio failures over kLOC and used as the dependant variable in a multiple linear regression analysis. The metrics were used as predictors.

To determine the efficiency of identifying high and low quality components correctly, a statistical normal lower confidence bound formula on the black box test failures was used. All programs having a black box test ratio of failures per 1000 lines of code lower than the calculated lower bound from the formula were of high quality and the remaining of low quality. The point estimate of the percentage correct classification was 90.9%, which means that 20 of the 22 programs were correctly identified as high or low quality.

Another similar research was made by Li [13]. Terms used in literature and the industry to describe software related problems include faults, errors, failures, bugs and defects. Faults and errors are for example when software produce wrong results. Failures can be that tests are failing or that the software fails to run. Bugs and defects is when something is wrong with the code resulting in faults, errors and failures. Some studies or companies define these terms differently. To avoid confusion, Li uses the term *field problems* to include all terms, with the requirement that the software problem occurs in the field. The term field problems is intended to be generic and to encompass all the terms used in the literature to describe software related problems in the field [13].

Li divides models that predict field problems into two classes: time based models and metric based models, by using an adapted classification scheme from Schneidewind [28] and Tian [33]. In his research he analysed metric based models in detail.

3.1.1 Time based models

Software reliability assessment is important to evaluate and predict the reliability and performance of software system. The models applicable to the assessment of software reliability are called Software Reliability Growth Models (SRGM). An SRGM provides a mathematical relationship between time span of testing or using the software and the cumulative number of faults detected. It is used to assess the reliability of the software during testing and operational phases [30]

Both time based and metric based models are SRGMs. Time based models use the problem occurrence times or the number of problems in time intervals during testing to fit a software reliability model. The number of field problems is estimated by calculating a number of problems in future time intervals using the reliability model. Time base models assume

that the software system has some probability of failure during execution. Thus, a problem occurrence is a random process in time according to Musa et al. [18]. The process is dictated by the number of residual problems and the discovery process, for example the amount of execution time. The idea is that every moment of execution has a chance of encountering one of the problems remaining in the code. The more problems there are in the code, the higher the probability that a problem will be encountered during execution. Assuming that a problem is removed once it is discovered, the probability of encountering a problem during the next execution decreases.

The main difference between different time based models are the model structures of underlying software reliability models. The important form of the software reliability models is the failure intensity function defined by Lyu as the rate of problem occurrence at a certain time [15]. The number of field problems is estimated by integrating the failure intensity function. The commonality between time based models is the use of time related problem occurrence information to fit a reliability model and then predicting field problems using the fitted model. There are many time based models and Farr discusses 17 different ones in [15].

3.1.2 Metric based models

The metric based models can predict field problems using metrics available before release that capture various attributes of the software product, the development process, the deployment and usage pattern, and the software and hardware configurations in use. Metric based models use historical information on metrics available before release, which are called predictors, and historical information on software field problems to fit a predictive model [13]. The fitted model and predictors' values for the current observation are used to predict field problems for the current observation. Because of the use of metrics, effects of various attributes on field problems can explicitly be accounted for in the models. The idea is that certain characteristics make the presences of field problems more or less likely. Capturing the relationship between characteristics and field problems using past observations, allows field problems to be predicted for unforeseen observations. A difference from time based models is that metrics based use historical information on predictors and the actual number of field problems to construct the predictive model. Different metrics based models use different modeling methods to model the relationship between predictors and field problems. Since there is no assumption about the similarity between testing and field environments, metrics based models are more robust against differences between how the software is tested and how it is used in the field [13].

3.1.3 Metric based inputs

The inputs to metrics based models are metrics' values. The metrics can be categorized and Li does this by using a version of the categorization schemes used by Fenton and Neil [6], Khoshgoftaar and Allen [11] and the IEEE standard for software quality metrics methodology [4]. The categories are *product metrics*, *development metrics*, *deployment and usage metrics* and *software and hardware configurations metrics*.

Product metrics measure the attributes of any intermediate or final product of the software development process [4]. Product metrics have been shown to be important predictors by studies such as Khoshgoftaar et al. [12], Takahashi et al. [32], Jones et al. [10] and Shelby and Porter [29]. According to our findings during the literature study of this thesis, these are the most used metrics in studies. Many of the product metrics are highly correlated and measure similar things, for example lines of code and source lines of code.

Development metrics measure attributes of the development process. These metrics are usually computed using information from version control systems and change management systems. The idea behind development metrics is that attributes of the development process, that is how the product is implemented, is related to field problems. Areas where metrics can be measured include problems prior to release, changes to the product, people in the process and process efficiency.

Deployment and usage metrics measure attributes of the deployment of the software system and usage in the field. There have not been many studies examining deployment and usage, furthermore no data source is consistently used. The amount of execution and the kinds of execution during operation are related to field problems.

The last category, software and hardware configurations, measure attributes of the software and hardware systems that interact with the software system in the field. Some field problems can only be exposed by using specific configurations, therefore the software and hardware configurations in use are related to field problems.

All of these metrics that are available before a release are potential predictors, which may be used to predict field problems and usually computed using information in change management systems and defect tracking systems. This type of metrics include the number of faults, bugs, errors and defects.

There is no right answer to which specific metrics to collect and in general there is an agreement on the need for more metrics that capture different attributes as stated in the IEEE standard for software quality metrics methodology [4]. Prior work shows that in general, collecting and using more metrics will result in more accurate field problem predictions. The general approach is to collect all reasonable metrics that are consistent for all observations within the study. Metrics which measure attributes that can be reasoned as being related to field problems and are measured in the same manner for all observations are also good. There are three ways of showing that a predictor is important. The first way is to show that there is a high correlation between the predictor and field defects. For example one can select all predictors that have a correlation higher than a certain number. The second way is to show that the predictor is selected using a model selection method. Here a linear regression model can be used to predict the number of errors for example. The p-value of the estimated parameter value can be used to select important predictors. If the p-value is low for a certain predictor then it is likely to be a meaningful addition to the model because changes in the predictors' value are related to changes in the response variable. The third way is to show that the accuracy of predictions improves with the predictor included in the prediction model. An example of this way is shown in Jones et al. [10]. They construct two logistic models that

classify modules as risky and not risky. One model uses only product metrics and the other uses product metrics and a deployment and usage metric. The authors show that the model with the deployment and usage metric has fewer errors of a certain type for the testing set and argue that since identifying risky modules (i.e. not making those errors) is more important, the model with the deployment and usage metric is better. Therefore, deployment and usage metrics are important.

3.1.4 Model outputs

The output is what comes out of the metrics models. Depending on the result different actions may be taken to reduce the cost and effect of field problems. According to Li, prior work shows there are three levels of output [13]. A level 0 output is a relationship between predictors and field problems. The research question addressed for a level 0 output is "what predicts field problems?". Studies that produce level 1 or level 2 outputs automatically include level 0. Studies that only report level 0 output simply observe a correlation between the predictors' values and the values of the field problems metric. A level 1 output answers the question if an observation will be risky or not. Prior work at level 1 establishes relationships between predictors' values and the class of the field problem metric using models, and then uses the models to classify observations as either risky or not risky. The output is thus a classification. The primary purpose of a level 1 result is to focus testing efforts on risky modules. Level 2 output answers what the number of field problems is. Prior work establishes relationships between predictors' values and the value of the field problem metric using a model, and then uses the model to quantify the risk. Results are predicted values of the field problem metric. Some methods that give level 2 outputs are linear modeling (linear regression and negative binomial regression), non-linear regression, trees and neural networks. Determining the number of field problems should be done meticulously. It may allow the appropriate amount of maintenance resources to be allocated. The output can also be used to determine where to focus testing by selecting a number of the observations. The most commonly used measures of accuracy for the output are the average relative error (ARE), the average absolute error (AAE) and the standard deviation of the absolute errors. The AAE measures the average error in predictions, that is how far off a typical prediction will be. The ARE measures the average percentage of error in the predictions, that is relative to the actual number of field problems, how far off a typical prediction will be.

3.1.5 Counter findings

There have been many studies in trying to predict post-release defects based on pre-release data and it is important to take their lessons to heart. It is important not to take every point as a matter of course, but to check every correlation, as shown by Fenton and Ohlsson [7], in which the releases they examined showed several counter-intuitive notions. For example, it showed that the modules that are fault-prone pre-release tended to not be the same modules that were fault-prone post-release. It also showed that fault-proneness were not strongly correlated to size, contradicting the notion that more code equals more faults. These findings by Fenton and Ohlsson do not discredit that there often are correlations on these metrics, but it does show that different system structures and release processes can show different correlations.

A more recent paper by Pecorelli [23] also shows some findings that might be counter-intuitive. "*The main surprising outcome of our study is that most of the considered test-related factors are not able to explain post-release defects.*" [23]. He actually found that the two terms "assertion roulette" and "eager test", which are indicators of a bad test case, were the only test related factors that had an impact. Furthermore they actually showed a negative correlation, indicating that a test case affected by these negative symptoms gave fewer post-release faults.

3.1.6 Goal Question Metric

To decide Q_{in} and Q_{out} we found that a popular method was the Goal Question Metric (GQM) approach [3]. The approach is based on the assumption that, for an organization to measure in a purposeful way it must first specify the goals with the project. Then it traces those goals to data that should define the goals operationally and provide a framework for interpreting the data with respect to the stated goals.

There are three levels in the GQM approach: conceptual, operational, quantitative. Firstly, in the conceptual (GOAL) level a goal is defined for an object. Objects of measurement can be categorized into *products*, *processes* and *resources*. What counts as products are artifacts, deliverables and documents that are produced during the system life cycle. Processes are software related activities normally associated with time, for example, specifying, designing, testing and interviewing. Resources are items used by processes in order to produce their outputs, for example, personnel and hardware.

The second level is the operational (QUESTION) level. This is a set of questions used to characterize the way that the assessment of a specific goal is going to be performed based on some characterizing model. Questions try to characterize the object of measurement, one of the four categories in the conceptual level, with respect to a selected quality issue and to determine its quality from the selected viewpoint.

The third and last level is the quantitative (metric) level. A set of data is associated with every question in order to answer it in a quantitative way. The data here can be either objective or subjective. If they depend only on the object that is being measured and not on the viewpoint from which they are taken, it is objective, for example number of versions of a document or size of a program. If they depend on both the object and the viewpoint, it is subjective, for example level of user satisfaction. Some of the data here we call Axis specific because it is produced by Axis. For example, data coming from their cameras.

3.2 Linear regression

If there exists an a and b such that $y = ax+b$, for every variation of x and y, then there would be a perfect linear relationship. This means that if we ever lack the value of y, we can simply look at the value of x, and calculate y. In general x is known as an independent variable, and y as a dependant variable and another word for this relation is *collinearity*.

Linear regression is a way of viewing the relation between two real variables, in a way that is similar to a linear relationship. True linear relationships are rare in reality, but many relationships can be simplified to fit such a relation. Especially if the range of x is short enough.

In reality, the sample points of x and y is not as clean as in the world of simple math. There are always causes of errors, be it hidden variables, faulty sampling tools or rounding errors. If there is enough data, it can be reasonably accurate linear relationship between the two variables. See Figure 3.1 for an illustration. When performing linear regression, one only has access to the sample data, and as such can only try to estimate the actual linear relationship.

In linear regression the observations, marked as green circles, are assumed to be the result of random deviations, the red lines, from an underlying relationship, the straight line, between a dependent variable y, and an independent variable x. The goal of simple linear regression is to create a linear model that minimizes the sum of squares of the residuals, which are the red lines in Figure 3.1

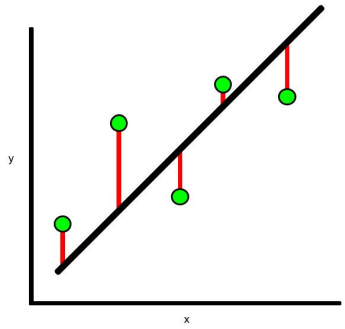

Figure 3.1: Simplified linear regression. The diagonal line represents the actual relation between the two variables x and y. the circles are the sample data that have been collected. The vertical lines are the amount of error that are included in the sample data

It is generally as such that x is assumed to be the catalyst that causes the change in y, and there thus is a cause and effect relationship between them. There is however no mathematical restriction that says that such must be the case. There may well be a hidden variable Z that has a direct linear effect on both x and y, and is the true cause of the change and linearity in x and y.

This means that without white-box knowledge of the system in question, there is no guarantee that the relation is causal. Even if the sample data were to perfectly reflect a linear relationship, it only does so for the given points of the data. If a new point of x were to be introduced, that were far from the other points of the samples. It is possible that it is so far from the other points that it has passed over a threshold where the relation is no longer representable as linear, but rather exponential as illustrated in Figure 3.2.

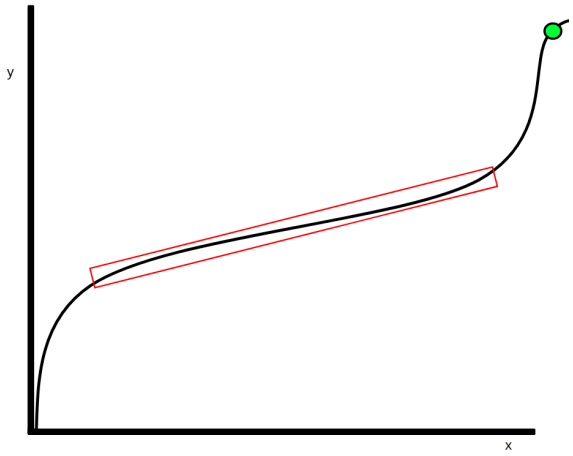

Figure 3.2: An illustration of how a non linear relation can be represented as a linear regression. As long our data is within the selected area, the relation seems linear. But if samples from the top right circle are included, our model gets thrown of.

3.2.1 Multiple linear regression

The case of one independent variable is called simple linear regression. For more than one independent variables, the process is called multiple linear regression. This can be seen as a merger of several linear regression models, and results in something on the form of $y = Ax_1 + Bx_2 + C$.

A multiple regression model is based on several assumptions. [2]

- There is a linear relationship between the dependent variables and the independent variables.
- The independent variables are not too highly correlated with each other.
- The observations are selected independently and randomly from the population.
- The residuals should be normally distributed with a mean of 0.

3.2.2 Multicollinearity

Simple linear correlation is a linear relation between two variables. If we merge two simple linear relations, we do not end up with only two relations. We also introduce a third relation between the two independent variables. Indeed, every new independent variable we merge into the multiple linear regression, creates a new relation to every other independent variable. The total number of relations in the model is in fact $n(n-1)/2$, where n is the number of variables.

Imagine that we are making a multiple linear regression on commuting time. We have made several simple linear regressions that have given good results, and its time to merge them.

So we try to merge two of our regressions, called alpha and beta, into the regression omega. Alpha showed a great correlation between distance in meters to commuting time, and beta distance in kilometres to commuting time. And when we merge them into omega we find a good model that predicts commuting time. The only problem is that the two variables are measuring the same thing. Furthermore, the coefficients of the multiple regression have almost an infinite number of variations that will fit the model. This means that any errors, as shown in Figure 3.1, will have a great impact on the model produced, which results in drastically different models based on slightly different sample data. The number of relations that can cause multicollinearity can be expressed by the same formula, except that n then stands for the number of independent variables.

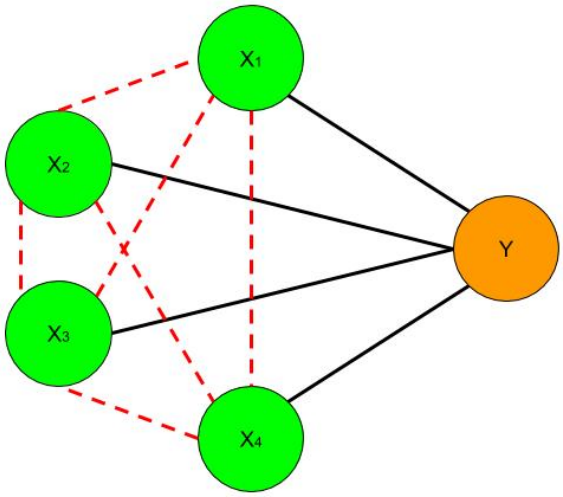

Figure 3.3: Multiple linear regression. The dotted lines are the relations that might cause multicollinearity

Adding more independent variables creates more relationships among them. Not only are the independent variables potentially related to the dependent variable, but also to each other. When that happens, it is called multicollinearity. The idea for a good model is that all of the independent variables are correlated with the dependent variable but not with each other. One way to solve the potential multicollinearity problem is performing a PCA, explained in the next section.

3.2.3 PCA

PCA (Principal component analysis) is a way of grouping samples according to properties. Often this is done because there are too many properties for it to be convenient or even possible to illustrate in a graph.

In practise the walls of different properties are broken down and new properties are created from the remnants. The new properties are denoted as PC1, PC2 and PC3 e.t.c., and are designed as such that PC1 is the property that shown the greatest amount of difference between the samples. The number of new variables that are created is the smallest of the number of samples and the number of variables, but generally one is only interested in the first two or three. This is because just like how the first PC variable shows the greatest amount of variance, the last one barely shows any at all.

With this grouping it is then possible to see hidden patterns in the samples. If one of the new properties is composed mainly by a single one of the original properties, it would indicate that said property is not strongly correlated with any of the other properties. Similarly if more than one of the original properties have a great impact on one of the new properties, it would indicate that these original properties are highly correlated and should not be used together.

3.2.4 Overfitting

Adding more independent variables to a multiple regression procedure does not necessarily mean that the regression will be better or offer better predictions. It could as well make it worse. That is called overfitting. Adding more variables will explain more about the variation of the dependent variable but it can lead to many other problems that you want to avoid. Instead of just adding more variables the idea is to pick the best ones for the model. Some independent variables are better at predicting the dependent variable than others. Some contribute nothing. Occam's Razor, says that when using models and procedures, to use only that which is necessary for the modeling but nothing more [9]. For example if a regression model with 2 predictors is enough to explain the dependent variable, then no more then the two predictors should be used. So overfitting is the use of models or procedures that include more terms or more complicated approaches than are necessary.

According to Hawkins overfitting can be split into two types [9]. Using a model that is more flexible than it needs to be, and using a model that includes irrelevant components. An example of the first one is using a neural net able to accommodate some curvilinear relationships, for a linear model. Since it is curvilinear it is more flexible than a simple linear regression. However if used on a linear model, it adds complexity without any benefit in performance. On the contrary, it can instead give worse performance than the simpler model. The second type is what it sounds like, for example a multiple linear regression that contains unnecessary predictors.

There are several reasons why overfitting is unwanted. Adding predictors that do not do any useful function means that in the future when using the regression to make predictions,

you will have to measure and record these predictors to be able to substitute their values in the model. This does not only waste resources but also increases the possibilities for undetected errors in databases to lead to prediction mistakes. In a feature selection problem, models that include predictors that are not really needed result in worse decisions. A mistaken decision to use a certain predictor in a model when this predictor is irrelevant leads to wrongly ignoring other compounds based on the irrelevant predictor. This way valuable leads can be lost. Adding irrelevant predictors can make predictions worse because the coefficients fitted to them add a random variation to the following predictions. Lastly, models impact portability. A linear regression model with one predictor that captures a relationship is very portable. It can be applied by anyone, anywhere to their data simply by knowing the two numbers, the values of the regression slope and the intercept. There are also models that are not at all portable, which can only be reproduced by reusing the modeler's software and calibration data. In science, a vital requirement is that other people and scientists can use your work or redo it in another location. With that said, more portable models are preferred to less portable [9].

3.3 Analytic Hierarchy Process (AHP)

The idea behind AHP is to prioritise what ever metrics or attributes you have. You start by comparing each metric with all other metrics and put a number of how much more important the first metric is compared to the second. You then proceed by calculating averages of the comparisons to get the weighted criteria. Lastly, a consistency check is made to make sure the numbers put on the importance was not just random but actually had some thought. A more detailed explanation follows.

First you create a matrix where each row and column is represented by a metric. This means each cell is represented by two metrics. How AHP works is that you start by comparing two metrics and simply put a number between 1 and 9 how much more important the first metric is compared to the second metric. With one being equally important and nine being of extreme higher importance. Then you continue by comparing the first metric with the third, and so on for all. When the first metric has been compared to the rest, you move on to do the same thing with the second metric. When comparing the second metric with the first metric now, since you have already done it once, it will automatically get the inverted value to the value you put when comparing the first metric with the second. For instance, say metric A is strongly more important than metric B, you put a 5 in the cell that is in row A and column B. When you later compare metric B to metric A, it will get 1/5 since we stated that metric A was more important and already had a 5. You do this for all the metrics which means you will complete a table, such as the one in Figure 3.4.

Metrics	A	B	C
A	1	3	9
B	1/3	1	4
C	1/9	1/4	1

Figure 3.4: AHP matrix after prioritizing

The next thing we want to do is get the weighted criteria. First you normalize the values by computing the sum of a column, and divide the value in each cell, in that column by the sum. This is to be done for all columns which gives a normalized matrix, see Figure 3.5 The average of the values in each row in the normalized matrix represents the weighted criterion for the metric at that row. So for the first metric, say A again, we take the average of all values in the first row, and that average is the weighted criterion for the first metric.

Metrics	A	B	C
A	0.7	0.71	0.64
B	0.23	0.24	0.29
C	0.06	0.06	0.07

Figure 3.5: Normalized matrix

Lastly we checked for consistency, that is, if the values put into the rankings were done with some thought or just randomly. To do this we calculated a consistency index. To make it easier to understand we are going to call our initial matrix without normalization $\boldsymbol{C}$ and the weighted criteria vector is going to be called $\boldsymbol{W}$. We multiply our initial matrix, Figure 3.4 with the weighted criteria vector to get a weight sums vector W_s: $[C]\{W\} = \{W_s\}$. With this we can find the consistency vector $\boldsymbol{Consis}$ by taking the dot product of our weighted sum vector and one divided by the weighted criteria vector: $\{\boldsymbol{Consis}\} = \{W_s\} \cdot \{\frac{1}{W}\}$. Lastly, we determine the average of the elements in the consistency vector and call that value $\boldsymbol{\lambda}$. Now we can calculate the consistency index $\boldsymbol{CI}$ as $\boldsymbol{CI} = \frac{(\lambda - n)}{(n-1)}$ where n is the number of criteria. There is one last step to determine if the comparisons were consistent. We divide our consistency index with a value called random consistency index to get a ratio: $\boldsymbol{CR} = \frac{CI}{RI}$. For n = 7 which is the size of our matrix, the RI is 1.32. If this ratio is less than 0.1, that is 10%, then we can say that the evaluation is consistent.

3.4 Black box testing

In the language of Verification and Validation (VV), black-box testing is often used for validation, i.e. are we building the right software, and white-box testing is often used for verification i.e. are we building the software right? [21].

With black-box testing the tester should not have access to the internal source code itself. The focus is on the output generated by the "black box", which is the code, in response to the selected inputs and execution conditions. The tester knows only that various data can be input to the box and the box will send something out. This can be done based on requirement specifications. Furthermore the tester knows what to expect as output from the black box and tests to make sure what it sends out is correct [21]. Black-box testing is done based on customers' requirements, so any incomplete requirement can easily be found and addressed later. Furthermore it is based on end user perspective. During black-box testing, testers need to be involved from customers' requirements gathering and analysis phase. In the design phase test data and test scenarios need to be prepared [21].

Our work is similar to black-box testing with a few differences. The most obvious is that we are not really testing. However, similarly, in our case we have a model, an input to that model, and it gives an output. There is no rule saying that Axis personnel cannot look in the code but as in black-box testing they are more interested in the output. When deciding inputs to the model we had discussions with our Axis supervisors who recommended some metrics in addition to the ones we found by ourselves. The same goes for the metrics used in the model to calculate the output. When evaluating the model we will use the output we get for the newer releases and compare it with given data in hope that they will be close. So this is the test to make sure it outputs what it should. In fact the comparison actually tells how good the model is in predicting.

3.5 Case company specifics

3.5.1 Software-hardware

One of the specifics with the selected company is that their products are a combination of software and hardware. Furthermore, each product in and of itself is a unique combination of different software and hardware components. This means that looking at just the software side is a sure way of not getting an accurate view of the product. The software for the cameras, is called firmware. This is a specific class of computer software that provides the low-level control for a device's specific hardware [1], where the devices are cameras. In other words, the firmware is what controls the parts inside the camera. Most products have different firmware. However, it works in the same way, controlling the hardware, and often different firmwares are released at the same time.

There is little in the area of software-hardware metrics in the literature. Those that exist are either specific to the area of their selected company, or they are largely not applicable to our product area, as most of these examples are based on personal computers and therefore use metrics such as operating system [16] [17].

3.5.2 Release branch

The current release procedure is that at predetermined points in time, a release branch is forked from the main branch. That main branch is continuously tested and is the version that produces most of the available test data. The release branch is also tested, but not as regularly as the main branch. As such, there might be new defects found on the main branch, while the release branch is still alive. These defaults may be present in the release branch, but not necessarily. As new code is continuously added to the main branch, new defaults can be introduced there.

At the branching point some high severity bugs receive a higher priority than others, as in, the release will not happen unless they are fixed. Any fixes that are made on the release branch is also applied on the main branch, and are thus relevant for subsequent releases as well. The opposite can also occur, high severity bugs might be downgraded, because although

a fault is severe, it might not affect the customer. However, if a new feature is implemented on the main branch after the release branch is forked, it will not be included in that release.

What is released is mostly updated software for existing products. New features, upgrades to performance and patches to remove defaults. New products and supporting software can also be released simultaneously.

3.5.3 Testing

There are several layers of testing at the case company. There are the most expansive tests that might span several releases. These are the most in-depth and take very long time to complete. Then there are the pre-branch tests that are done before the release branch is forked. These tests are more designed to test vital functionality without taking as much time, as to be possible to run between releases. Lastly are the RC (release candidate) and pre-RC tests that are run on the release branch. These are the fastest tests and are designed to be fast. Depending on the development of the release, several of these test runs might be run. There is no pilot or customer testing before release. The feedback that comes after a release is then considered for the next release.

3.5.4 Release data

There is historical data for a total of 20 releases, this is a fairly low amount when creating a complex model. Preferably, We also separate some of the releases for model confirmation, further decreasing the number of available data sets. As such we must limit the amount of metrics as to avoid overfitting, so that our model will not become volatile.

Like any software company, the workflow at our case company have changed over time. As the company have expanded, there have been new products and departments introduced. The workflow has been refined and standards have been introduced that were not there at the start. Different tools have been introduced and discarded, and some new tools have not been used for very long. Data that were collected have been discontinued as it were eventually deemed problematic. Furthermore, some very useful data have begun collection very recently. This all means that even if we have access to 20 releases, the information that is available is highly varying. Some metrics that are logically highly interesting may only be available in the last 3 or so releases. This means that showing a significant correlation will be near impossible.

Chapter 4
Method

This chapter is partitioned into 2 parts. The first section, Section 4.1, contains a presentation of the thesis work in a temporal order. The second part, Section 4.2, provides a more in-depth exploration of the choices and work that was done. It is also in this chapter that some of the more complicated concepts are explored.

4.1 Research method

This section describes our entire work-procedure to create the models and evaluate them. The overall method used was a case study [26]. We have organised subsections in the order that they occurred. As a way to summarize the entire procedure, our workflow is further moved into context here with Figure 4.1.

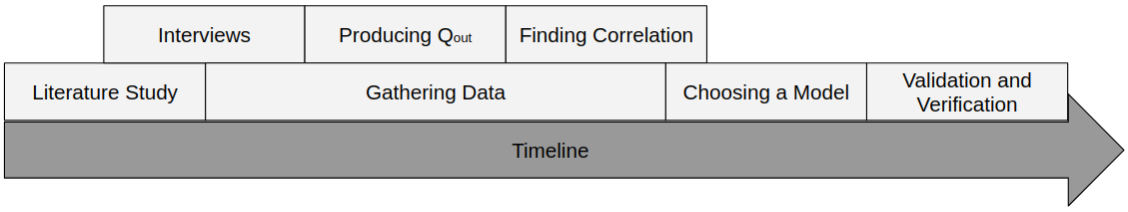

Figure 4.1: timeline of our workflow, and the structure of this section.

1. The literature study section gave us an understanding of the subject and a language to use.

2. The interviews section gave us an understanding of the company and what metrics could be available.

3. The gathering data section provided us with the data for both Q_{in} and Q_{out}.

4. The producing Q_{out} section is about generating a way to quantify quality, so that we have a time-series to run our other metrics against. So the output of this stage was a single time-series that described the quality of each release.

5. The finding correlation section divides what metrics we have into those that have a significant correlation and those that do not.

6. In the Choosing a model section, a functioning model is produced.

7. In the final section, Validation and verification, there is an evaluation of the models and the work that was done.

4.1.1 Literature study

We started our work by doing a literature study. This allowed us to learn more about the subject and gain a common language to use within the subject. At first we only read abstracts and skimmed through the found literature. The field of literature we searched in was primarily software quality. Our top used words in search strings were *software*, *release*, *metrics*, *quality*, *development* and *delivery*. We found approximately 50 articles and used half of them. The title of the article, book or other paper was written to a document along with a link to it, comments about it and a grade from one to five how relevant it is. We kept all references in this document so we could easily access them when needed. When studying the literature we tried to find metrics that other researches had used and methods of how to measure quality. As with the literature, the metrics found were collected and written down in a separate document for future reference. To synchronise our work and be more aware of each other we also wrote down the search strings to not do things twice. At this point we were only trying to find literature and not really read it thoroughly. Later we would go on with reading the most relevant ones, both of us.

4.1.2 Interviews

Interleaved with the later part of the literature study we began to have interview meetings with personnel at the case company. Our supervisors at the case company are highly involved with the release process and were thus able to get us into contact with various other departments also involved in the release process.

The interviews helped us to learn more about the respective departments, for example if there are any in-house metrics we could use. We structured it as such that it would be mostly a free-flow discussion. In each, we explained what the thesis is about, the goal, and what we need. The personnel were all seniors that came from different departments and have different roles. The departments were product management, QA, tools and data diagnostics management. The roles were amongst others project leaders, test leaders and product specialists. In the end we had held about 7 interviews with different levels of formality. The results of these interviews is the information that can be found in Section 3.5.

We had two main questions, which were: "what tasks are you performing?" and "what data do you work with?". If the data seemed relevant for us we would also get access to it. The data

could either count as Q_{in} or Q_{out}. Data from different departments were stored in different places, for example different Elasticsearch clusters, databases or other text documents. We wrote down the bulletpoints and a few specifics from the meetings on paper and later also on an internal wiki page on Confluence. Furthermore, the interviewed personnel sent emails with links to the data as well as other persons of interest that might be able to help. Most of the people directed us towards the "IDD" department which is supposed to have "external data". We eventually did a final push to reach out to new departments and people to try to find additional resources. We found that they only directed us to people we had already talked to and resources we already had.
Once we found ourselves towards the end of the initial literature study phase and started on the interview meeting phase, we ranked the metrics found in the literature study. We did this firstly by both of us independently giving every metric that we had found a coloring in the green-yellow-red spectra. The green spectra meaning "good and/or is likely to be available", and the red representing the opposite. We then merged our two rankings by explaining our individual reasoning. At this stage our understanding of the inner working of the case company were only starting to be fleshed out, so the rankings were left with several metrics that we were hopeful they would be available, and if so, they would be interesting to look into. Furthermore, all the metrics included in this ranking were metrics found in the literature study.

When all interviews had been done and we were not directed to any new department or personnel, we did a re-ranking of our metrics. This time we used the same colors, but with slightly more distinct meanings. Furthermore, we had gained some in-house metrics from the interviews that we did not have in the first ranking which was after the literature study. Green were metrics that were either available or the product of processing available metrics. Red were metrics that were not available and/or not noted as important. Yellow were metrics that were deemed interesting but not directly available. These were metrics that we would suggest to the case company that they should collect if they want to continue this line of work as well as the metrics that we deemed possible to extract with a lot of work. It might be that they have a good correlation, but as stated by Peng and Wallace [24], on the properties of a good metric: *"Easily collected - the cost and effort to obtain the measure is reasonable."*. All the data we extracted was done through python scripts. This required the data to be either in some database that could be accessed through python, or readable files. If it wasn't, then we would not be able to extract the data fast enough with our limited time and resources.

4.1.3 Gathering Data

As illustraded in Figure 4.1, we now began to extract data. Since the data, both Q_{in} and Q_{out}, was spread out and not directly usable from its source it had to be extracted and processed. For this we wrote several scripts in python. Since the scripts did not have to be too advanced we split the work, that is, we did not do pair programming. However, we shared repository and used each other's methods if needed. We used a script template defined earlier by Axis and then added what ever our scripts were supposed to do. To extract the necessary data we had three different ways because of the data being spread out. The first way was writing to an API which talks to the ticket-system holding all tickets. There were some problems using the API. It did not always show, visually, the same results as the actual ticket-system

which forced us to do cumbersome checks to make sure we had the right data. Furthermore, the amount of tickets we wanted to extract was large which the API could not handle and crashed. To solve this we had to make many more smaller searches and then sort these to not overload the API. The data was returned in json format which then had to be processed and written clean to files. The second way was by communicating with elasticsearch since some data is stored in elasticsearch indices. Luckily python has good support for this. The data here was also returned as json and had to be processed the same way. The last way was simply by copying hard data from old reports.

4.1.4 Producing Q_{out}

In order to do any correlation we needed to define our Q_{out}. This was a long process described in more detail in Section 4.2.3. We needed to reflect the quality of the releases using a reliable oracle that was available for all the releases. We had preferred to ground the quality in customer feedback or employee knowledge but it became apparent that this data was not entirely suitable. We ended up using employee knowledge to decide the relative importance of a selection of negative quality aspects. We did this with a second set of interviews. The results of which can be found in Section 5.1. This meant that our Q_{out} became an indicator of bad quality release, as in a high Q_{out} value indicates a bad release.

The data for each release was extracted in the same way, with a python script. The only difference in the script between each release was the start date and end date in which the data should be extracted from. When the data was extracted, the values were just put in the Q_{out} formula and we had a number.

4.1.5 Finding correlation

After acquiring a quality numbering of the past releases (Q_{out}) with a quality method, we had a proper pattern to use in our correlation modeling. We then did a structured correlation checking between all the Q_{in} we had in order to check correlation between the Q_{in} and our assigned Q_{out} values. These values were computed through a python script and produced to a spreadsheet that showed the correlation, as Pearson value, between each Q_{in} and Q_{out}. We also had a critical value which is based on how many data points there were for that Q_{in}, a standard significance level of 0.05 and is the minimum that the Pearson value can be in order to say that there is a correlation with significance. We used this as ground to start our multiple linear regression correlation analysis.

At the time of the second status meeting for the thesis, we were at the position of having gathered all the data that were available as either Q_{in} or Q_{out}. We had gone through the process of gathering coefficients for the Q_{out} by way of a survey and AHP, and had produced data points with said data. We had also tried to gather all of the Q_{in} and categorised them into one of three categories:

- Impossible or too inconvenient to extract
- Not enough data points available

- Extracted

With the data that we had extracted we started to run correlation checks against our produced Q_{uot}, and were encountering some new problems. The main problem was what we refer to as the "null problem", more in Section 4.2.2, which is actually a collection of similar problems. Depending on the nature of the metric it is sometimes unclear if a zero signifies the value zero or if it means that the data is not available.

After choosing to limit our scope in terms of data points as described in Section 4.2.2, we recalculated our correlations that we had done and continued to analyse our data as described in Section 4.2.4. Once we were done with the things described in that section, we had a total of 8 metrics that we felt had potential to be used in a final model. These are those metrics that survived that procedure:

- Number of CST bugs
- Number of tickets with showstopper severity
- Number of packages from final to final with simple comparison (positive correlation to Q_{out})
- Number of packages from final to final with simple comparison (negative correlation to Q_{out})
- Memory available
- Boot-time
- CPU usage
- System ready time

We decided to use these as Q_{in} for the final model.

4.1.6 Choosing a model

Since we only had 8 metrics that cleared the previous steps, it was not at all inconceivable to test every model we could make and simply chose the one that showed the best numbers, and in broad terms this is what we did. This is explained in greater detail in Section 4.2.5, but in short we did the following:

We checked the ability of our models by removing 1 of the data points from the data and training the model on the remaining data. We then used the Q_{in}s of the removed data point to make a prediction of what the Q_{out} value would be. As we also knew the actual Q_{out} of the removed data points beforehand, we could now compare the prediction with the real value to confirm how good the prediction was. This was all done in a python script written by us, that simply iterated through a table of data, where each row represented a release and each column is a metric.

The result of this stage is a total of three different models, each with their owns strengths. The models all use different combinations of the following 4 metrics:

- Number of CST bugs

  This is the number of customer support tickets reported as bugs to the master project, i.e. what is supposed to be released. We measured this by counting all the tickets with a bug tag in the time interval between two releases. For instance, as soon as release 1 has happened, and until release 2, those tickets belong to release 2.

- Number of tickets with showstopper severity

  This is the number of tickets that has the severity tag showstopper. There are four severity levels, where showstopper is the worst or more critical. These tickets don't have to come from customer support. We measured them in the same way as with the bug tickets, in between two releases.

- Number of package changes from final to final with simple comparison (positive correlation to Q_{out})

  This is how many changes there has been in the packages from one final release to another, just as in the case with the tickets. A package is a folder containing code files to be used in the project. We measured this by iterating through a file for each release, containing package names and versions, and counting differences from the previous release. Simple comparison means that we just count every change as equally worth. I.e. if there exists a package for release 1 but the same package does not exist in release 2, that is a package change. In the same way if it does not exist in release 1 but is added to release 2, it is also a change. The last thing we count as a change is if it exists in both releases, but has a different version in each. This metric does only take into account the changes in packages that had a high positive correlation with Q_{out}.

- Number of packages from final to final with simple comparison (negative correlation to Q_{out})

  This is exactly the same as the one above, with the difference that this metric only takes into account the changes in packages that had a high negative correlation with Q_{out}.

4.1.7 Validation and verifying

To validate the models, we ran them on the current release, which we had not used in any of the previous work. This was done with a script that we wrote as part of the handover of our work to the case company. The script was simply a merger of the bits of the code that we had used to gather the metrics in the earlier steps. There is however one bit that was different with this script, which was that one of the systems that we communicated with had been discontinued by this point. So we had to gather those metrics through the new system, which it seems we managed to do in a way that is equivalent to the old one.

What we had wanted to do was to calculate the Q_{out} for the release as well. However the new release was rescheduled to be released some time after we left the case company. As

such we could only look at the predictions and the work that had produced them when we validated the results.

When it comes to the verification we once again wanted to ground it in the case company. To do this we presented our findings to a focus group of the employees that might be subjected to our work in the future. These are same ones we have had interviews and discussions with before, i.e. project leaders and product specialists. We invited them all to a meeting to get their opinions and thoughts. Specifically, we wanted to know if they had any use of the models and if the results were good enough. We were also interested in whether our work was something they could continue working on.

4.2 Design Choices

4.2.1 Metrics selection

Is size important?

The logical first thought is that of course the size of the release matters. The bigger the release is, the more things can go wrong, more bugs can be found, they might be larger ones. Furthermore classes or modules depending on each other increase the risk of defects since there probably are more of them in a bigger release. These are just some possible outcomes with a bigger release. However, it does not necessary have to result in a bad release because of that.

According to Ross [25], the size of the software system matters for development cost, schedule and reliability . But our model only accounts for what is in the release and not the whole system. The case may be that the size has greater significance when a group of specific metrics are chosen as Q_{in} and less significance with other metrics. It remains to be seen what importance the size of a release has for the model.

As for the literature, the finding of size as an estimator have been varied. Many, if not most studies we have found have used size in one way or another and there are several subgroups of metrics here, that are all rather correlated. Such as size of the whole software, size of different modules and size of changes. This includes metrics such as LOC (lines of code) and code churn (changed lines of code). Then there are some very closely related "defects by size" metrics such as defect density.

Different sorts of projects, by different companies, with different practises, in different studies, with different objectives have shown different results. Whilst most of them have shown a positive and useful correlation, others have found them to not significantly impact the result. Graves et al. for instance mean that the size does not improve predictions, *"In addition to size, other variables that do not improve predictions are the number of different developers who have worked on a module and a measure of the extent to which a module is connected to other modules"*[8].

On the use of tickets

In 2005 Li et al. tried to find field defect predictors for open source software [14]. Open source is software that is available to the general public, both for use and modifications. Anyone can be a contributor and submit work, but only trusted people can authorise changes. Said authorisation can be gained by showing continuous good quality contributions. They investigated a total of 139 different predictors using 3 different statistical method and found that the best indicator was the activity of the mailing list of the technical department.

Tickets are a promising source of metrics for us as well. Finished tickets can be used in several ways. The first is as a simple numerical counter. This can be further improved by the fact that tickets are given a severity, which will provide a higher degree of nuance. They can also be used to calculate a "Mean Time To Repair" (MTTR), which could be an indication towards any of the following:

- The complexity of the code was high, so it took a lot of time to address the ticket.
- That the department was overworked and had little time to spend on the ticket, thus indicating poor quality.
- That they spent a lot of time and effort to give a high quality solution.

Based on our literature study, the correlation is either positive or negative. Active tickets can also be seen as a counter of open defects. With the aforementioned severity categorisation, this should be a good indicator.

Quantifying quality

Finding values that are quantifying the quality of the releases can be hard. Before the release, tests and measurements produce various results, for example CPU usage or number of tickets amongst others. But after the release these metrics are not really measured. In fact they are, but on the master branch. The problem here is that the master branch and the release branch can differ, which disables the option to just measure before and after, since the circumstances are not fully the same and affects performance and quality.

So, even some time after a release, how do you quantify the goodness of it? The first contender is customer feedback, but as previously stated in a plethora of literature, you only get customer feedback when things go really wrong. However, it is something to work with and one can see the situation as if there is no feedback, then obviously, the release was good. The problem with this type of customer feedback is that it is in text format. This makes it difficult both to put a quality number to it, and to do so automatically.

As our model is supposed to be automatic, it can be difficult to utilise the fairly non-standardised customer feedback forms. This should not show itself too troublesome in our work, as it is something that just needs to be done once per release. But if the work shall be continued by the case company or future theses work, then any release that is added to the data-set will require this work to be done manually.

Another type of customer feedback that is a little easier to work with are bug reports. They get in the system as tickets which can be tracked and easy to manage. Alas, the crucial information is texted and it requires some additional work to extract the quality aspect of it.

The most promising source of data is from customers that have agreed to share statistics, known as external data. The statistics are being measured after release and sent back to Axis making it great to use as validation metrics. Currently Axis focuses heavily on memory available but other metrics can also be fetched if needed. A somewhat promising metric found in literature is SWDPMH (Software Defects Per Million Hours). This might be possible to estimate, since it is known how many cameras uses a certain software. The only problem is that most sources only report hardware bugs. This is not because they cannot report software problems but because hardware bugs are more frequent and dominant.

4.2.2 Null problem

An example of this problem can be a certain tag in the ticket system, that seems to have fallen out of favour for a while. On our second data point, the tag was used 2 times, then only appeared sporadically for the next 8 releases before coming into regular use after that. Furthermore this is a tag used to differentiate a type of ticket, that we have confirmed with the people involved, to have been used throughout all our data points. This means that we cannot completely trust the values that we have extracted to contain the whole truth, especially if it is a zero.

Our metrics are collected from different systems, and in turn, those systems have been used for different amounts of time. Meaning, our metrics have different amounts of data points available. We know for a fact that some of our metrics are not available before 2017, and thus know that those zeroes before that definitely signifies null rather than zero.

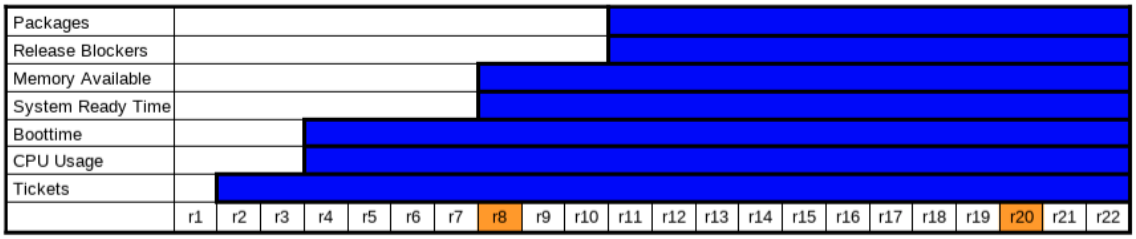

Figure 4.2: Data availability, an illustration of when the different metrics started being collected and thus what we have available. The marked releases are those that we had to manually alter the value of Q_{out} for.

As our regression will need to have data points for all the metrics that it is comprised from, we will need to either throw away data points or throw away metrics. The availability of data is visualised in Figure 4.2. Since we already are low on metrics, we decided to throw away data.

There are two additional factors that are to be taken into account in this choice. The first is that practises change, the work flow of the case company have changed. This implies that the older data points may describe a company and practises that are quite different from that of

the later data points. Which may well have a bad impact on the quality of our model. There are also two releases that we have manually changed the Q_{out} for. These releases were both instances where there was a problem with the "pulled products" variable. The problem was that the products that were pulled were not pulled due to low quality, rather there were other mediating factors. This means that they did not imply a lack of quality in the release, and if we did not manually correct these numbers they would heavily impact our Q_{out} falsely for these releases in a way that we knew was not real and true. At the same time, it is not sure that completely removing them correctly mirrors the truth. Some of these products might have been removed from the release either way, or would have resulted in several CST tickets. As this variable got a very high impact based on our Q_{out} coefficients, this is an important issue to raise. These are represented in Figure 4.2 as the orange releases.

We did not have many data points to start with, and now we have to contemplate whether to remove the first half of them, as well as an additional two, depending on which of the metrics we want to use.

As further described in Section 4.2.3, we ended up discarding the older releases. This decision was made based on two main arguments. Firstly, the continued development of the company is causing the system that is being modeled to change. As we are interested in the current version of the case company, the older data might have become misleading. Secondly, in constructing Q_{out} we want to take as many aspects into consideration as we could.

4.2.3 Constructing Q_{out}

When Modelling Q_{out} our main source of data was CST tickets, i.e. customer support tool tickets.

What is a CST ticket?

We had to define which tickets were actual CST tickets and which tickets were not. This was a problem because of the different result we got when using the web browser version of trouble and when using the API.

When using the web browser version and searching for "CST" it made a smart search, giving results that had the letter combination "cst" in any field, irrespective of casing. These ticket-groups are presented here in order of relevance.

- There are the results that have the CST tag, and the fact that we wanted these were under no debate. These were very easy to decide on as the tags are standardised and needed no interpretation.
- We also got results that had CST in the description, these were of a bit of a debate and sub-categorisation. The large majority of the cst-in-description tickets were tickets that should have had cst in tag, but the writer had forgotten to put it there. So we want to include most of these.
- For some results, CST was included in some other fields, but after looking through the tickets we found, it was clear that the vast majority of these tickets had CST in either

the tag or description. So we came to the conclusion that looking at these other fields separately gave no significant impact.

- The last group of results were a bit different, these are the false results. As the search is "smart", it also finds the letter combination inside of other words. This presented itself in the form of finding results where a certain variable "ExecStart" was referenced. This would be fine if they came under the same category as the previous results as well, but these had nothing else to do with CST that the fact that they contained the "cst" letter combination.

We used this web version when choosing what type of tickets we wanted to include, as it came with a ready interface and gave consistent results. It also allowed us to make sure that we did not have any bugs when we wrote our program that would work with the API of the system.

As there was no smart searching in the API, there were some inconsistencies when we wrote our program. We had to search for distinct results inside distinctive fields. We could of course search for only CST-in-tag, but then we would lose the CST-in-description results. That would not be good as this represented a respectable amount of tickets. We could also write code that scanned the description field for different combinations of the lettering "CST", that would include the results that we wanted, and exclude most of the false results. The only problem with this is the plethora of variations that people used to reference a cst ticket. In the end we came up with the definition of either having "CST" in uppercase in the tag or in the description, or having a link to a customer support ticket inside the definition.

Defining a number from tickets

When we started the work of collecting all the varying sources of Q_{out} that we had, we ran into the problem of turning the tickets into data points. The simplest way to use the CST-tickets to give a numbering would be to simply count them, but that would ignore a lot of the data available. Only tickets coming to the master branch, that were not fixed yet, were looked at. The tickets are divided to several criteria, being *type*, *severity* and *priority*. Type is divided into three different categories: *Bug*, *Feature* and *Change Request*. *Severity* and *priority* are divided into 4 different levels. For severity they are: *Showstopper*, *Serious*, *Medium* and *Small*. For priority: *High*, *Medium*, *Low*, *Incoming*. There is also the aforementioned effect of pulling products from the release scope and how time has an effect on the amount of CST tickets?

We gathered the data into a spreadsheet and made some Q_{out} graphs, making observations on what happened if different factors were changed. We gathered a greater understanding of the collected metrics and started to question certain aspects of them. Such as:

- White noise
- To what release does a ticket belong?.
- The divide between Q_{in} and Q_{out}

White noise would be the tickets that do not belong to the release under whose time period that they appear. For instance, that the ticket in question concerns a release which is two or three iterations back, rather than the intended release. The logical distribution of release ticket arrivals would be an early spike with a never ending exponential tail. Even if all the problems of a release are eventually mended, there could still be new CST tickets if customers upgrade their supporting software or platform, meaning that tickets could come even years later.

The time period, second point, intersects with the white noise part, as this reflects how these ticket arrival distributions could intersect. If two releases are close enough, then many of the tickets that belong to the first release will arrive after the second release has been shipped, essentially making the first release look better and the second worse than it is. There is also the bit where the white noise will provide a constant negative quality indication, that will accumulate more the longer a release interval is open. Meaning that a good release that is open for a long time can accumulate enough white noise tickets that it would get a worse score than a release that is actually much worse, but was only out for a little bit.

To combat these problems we started looking into ways to model the chance that a ticket belongs to a certain release. We looked at what happened to the data when we used different constant length intervals after release. We also started to work on finding the distribution shape of these intervals and ways to use probability coefficients to allocate problematic tickets to two releases. Luckily a large amount of these problems were solved at the beginning of the interviews of the long-term personnel occurring in Section 4.2.3, the results of which are found in Section 5.1. During the first interview we came across the subject and were informed that customer support was only offered to the current version and the LTS(long term support) versions. This does not completely remove the effects of the LTS originating tickets which can still interfere with the regular release tickets. However as we are able to filter on tickets that show up on the main branch, all the tickets we receive are relevant to some degree. The exception to this could be the tickets that are found on the LTS version but are deemed to also be of relevance to the main branch. These tickets would not be found on the release of interest, but the underlying problems that caused the ticket are still present in the release.

As for the case of release 1 tickets showing up in release 2, most of these are also fixed by supporting only the current and LTS versions. There is still a case that a CST ticket reported late in release 1 goes slowly through the customer support procedures and arrives into the internal ticket system after release 2 have been released.

The divide between Q_{in} and Q_{out} is, of course, also interleaved with the two other points. But simultaneously, it reflects how the data is actually continuous while the releases are more discrete in nature. Making the distinction of whether a certain variable is Q_{out} for release 1 or whether it is Q_{in} for release 2 is a highly complicated question. An example is the CST tickets that are marked as bugs. These were found by customers on the product of release 1, but are fixed between release 1 and 2. This means that this metric could be used as Q_{out} for release 1 or Q_{in} for release 2. Another example is pulled products. Depending on when the model is used, pulled products can be used as both Q_{in} and Q_{out}. If the model is used

before the decision to pull any products then it obviously can not be used as Q_{in} and the value for pulled products will be lower. However it can still be used as Q_{out} since one can see after release how many products were pulled which affects the quality. If the model is used after the decision to pull products, then it can be used as Q_{in} and will presumably give better results since it will match with the Q_{out}.

Pulled products

As seen in Figure 4.3, there is a case for using a "pulled products" metric when calculating the final quality for a release.

The scenario is, there are several problems during the release phase. Several of the products are deemed so unreliable that it is decided that they shall not be released. On release day, only a fourth of the original product releases are actually made available to the customer. The customers receive the good products that are reliable and they encounters few faults with their products.

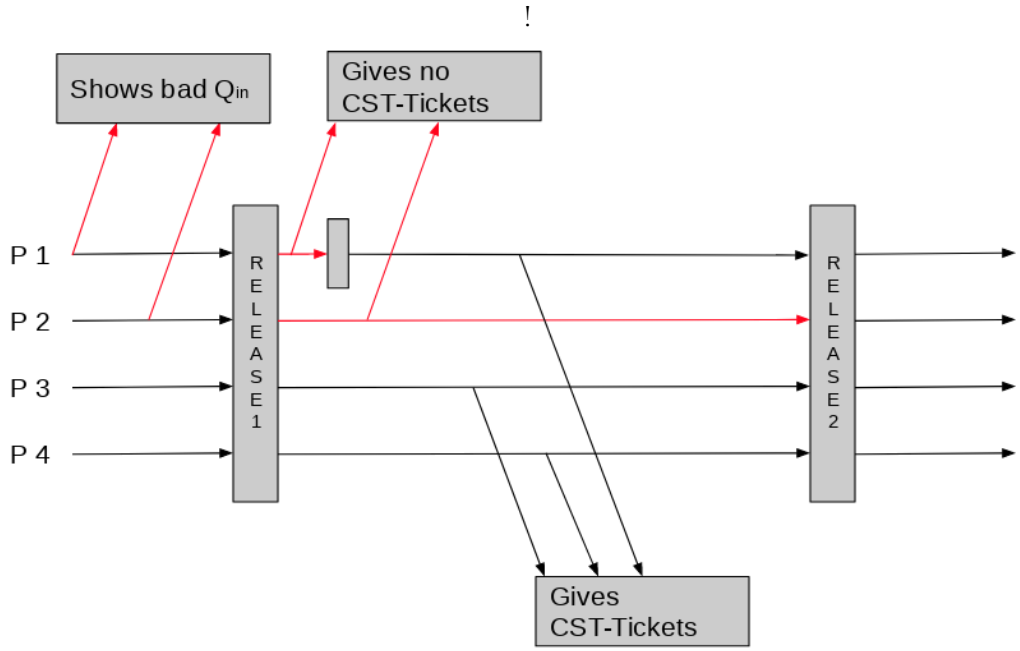

Figure 4.3: A figure Illustrating why using a "pulled products" metric might be a good Idea. It show 4 different products(P) within a release. P3 and P4 are deemed good and are released according to plan. P2 is deemed so bad that it is not released. P1 is also bad, but is quickly patched together and released a week later.

In this scenario, most of the Q_{in} will be foreboding of a bad release. But because appropriate actions were taken, there will not be a large amount of negative customer feedback. Despite the released products having a good quality, every metrics from within the company would be showing a bad result. It could even be argued that it was a bad release, even if the customer knew nothing of it.

Determining quality coefficients

We now had values for all the metrics that could make up the Q_{out}. We started making graphs and possible Q_{out}s by applying simple math to the metrics, such as adding the values together, multiplying with factors representing significance and occasionally dividing with different types of sizes. We started to individually process the data we had available, to find different ways to assign quality numbers to releases. We did this as to not influence each other and thus possibly find more options.

It was hard to decide on the final version of Q_{out}. There is no predefined quality formula or theorem so we had to figure it out on our own. We finally came up with the Q_{out} being the sum of the metrics. In addition, the different metrics and sub-metrics were to be multiplied by coefficients, as follows: $Q_{out} = C_1x_1 + C_2x_2 + C_3x_3 + C_4x_4 + C_5x_5 + C_6x_6$, where the x_i represents one of the metrics below and the c_i representing the coefficient to multiply with:

- Products planned for the release
  This is how many products (cameras) were planned to included in the release.
- Number of pulled products
  With pulled products means the products that were planned to be included in the release but in the end were not included.
- Delay
  Some releases are past their due date. This metric counts how big the delay is in days.
- Number of bug tickets
  This is the number of tickets that have the bug tag on them. Tickets can have different tags to help categorize them.
- Number of tickets by priority
  Similar to the number of bug tickets, but this metric counts the number of tickets by the priority tags instead. There are four priorities: incoming, low, medium, high.
- Number of tickets by severity
  Counts the number of tickets by severity, juts like with priority. Severity also has four severities: small, medium, serious and showstopper.

Initially there was also the metric "total number of tickets", but we realised that we were over representing tickets as a contributor, and total tickets was the one that gave the least amount of insight. That is why said metric is listed in the survey but not here.

When it Comes to the data collection scope of the metrics here, most of them are simply statical properties of the release, such as delays. But for the counting over time metrics such as bug-tickets, we mirrored the logic of Q_{in}, counting all occurances until the following release.

We wanted to let people at the case company be the input for the coefficients, as they know best what metrics that are more important than others. In order to be objective and systematic, a structured way of collecting these coefficients was needed. Our answer to this was a

method called *analytic hierarchy process (AHP)*[27]. This is described in more detail in Section 3.3. We proceeded to holding meetings with members of the case company where we tried to gather knowledge that could further help us decide on a Q_{out} calculation. One of the ideas of the meetings was to get professional and subjective input on the quality of the past releases. When we had done this, we could accumulate the results from the meetings, derive what Q_{out} metrics are important, and give a good quality numbering to each of the past releases. The interviews had 3 different parts:

- Free discussion
- Going through the old releases
- A survey

We structured it as such that it would be mostly a free-flow discussion. In each, we would reiterate what the thesis was about, the goal, and what we need. In order to bring structure to the interview, we would go through the releases whenever the conversation stagnated. This allowed us to take notes on specific releases as well as getting a broader context to their answers. We asked about their opinions and comments for each release and wrote down what was said and looked for things that were mentioned by everyone or specific opinions that stood out. The personnel came from different departments and had different roles. The personnel we had interviews with consisted of project leaders, test leaders and product specialists, just like the interviews at the start of the thesis. In total we had four separate interviews. The results of these interviews can be found in Section 5.1.

As we somewhat expected, people have forgotten the earlier releases, it varied between people and there was a trend of people that are working closer to the release-context remembering more releases. But from the people outside platform management, we barely got any clear answers for more than 3-4 releases back in time, as well as one of the older releases that were unusually bad. Our main gain from the semi controlled discussion was confirmation that the old bad release was bad, as well as context on the latest releases.

The more concrete data was gained through the surveys. These allowed the participants to take their time and think about the releases in general, rather than specific memories from any release. We ended the interviews by giving the participants our survey. We gave them some time to think and fill out the survey in their own pace. We still went through the poll to explain the various concepts and give context to the different questions, so that all participants understood the structure of the survey. The survey was based on the Q_{out} metrics that we had found, and was about the comparison between them. The answers of this was supposed to be interpreted through AHP, as described in Section 3.3. The basic structure of the poll was a pairwise comparison between each metric. We implemented this as a slider between each metric, as to make it as simple as possible for the participants. The poll itself can be found in the appendix.

AHP was applied on the results from the polls and collected. Based on the priority and consistency ratings that were produced, we then created a single unified priority rating and used that as the coefficients. When merging the different poll results we used the following

formula:

$$P = \sum \frac{P_i}{C_i^2} \tag{4.1}$$

This formula is of our own devising and is primarily based on the bad consistency ratings we got from the surveys. Here, P_i and C_i stands for the individual priority and consistency rating. When deciding on how to merge the poll results, we faced a dilemma:

- Most of the consistency ratings were far worse than what is expected.
- There is not enough data to discard any of it.

As to keep all the data, the less reliable data had to have a much lower impact on the final result than the data that was good. We came up with the solution of squaring the critical value divisor.

Once we had merged the poll results we normalised the resulting prioritisation so that they added up to a hundred. This was as to make the numbers more understandable and be consistent with the original AHP prioritisation, which adds up to a hundred.
Another way of defining Q_{out} that we had in mind was some sort of a boolean saying just good or bad, or high or low. The question would be what to base the prediction on. We did not feel like there was any reliable metrics that could be used to accurately tell if a release is good or bad. A tweak to this idea is to first compute a number like we decided on and then say if it is good or bad. But what would be the threshold, what number does the sum has to exceed to be good or bad? That is also something that we would have do determine ourselves. An idea is to compare it to previous releases and see if it was better or worse. But that does not necessarily tell if it still is good or bad.

We did not do any division by size for Q_{out}. Firstly, size can be anything, remember that we are talking about releases. What is the size of a release? Is it the number of products, lines of code, number of files, packages, the list could go on. Instead we decided to use size as input, not as a metric itself but as package changes and number of tickets. Furthermore, when calculating the Q_{out} we included number of products in the sum. It is the quality for a release that we are after and not the quality per products or any other ratio.

4.2.4 Constructing Q_{in}

Product metrics

There are statistics relevant to cameras that are pertinent to many embedded systems, such as various resource usages and booting times amongst others.

- Memory available
- Boottime
- CPU Usage

- System ready time

One of the most voiced metrics at the case company is memory usage, which is understandable as memory is one of the most expensive hardware components. Embedded systems are out in the field, and supposed to run by themselves for a long time on limited hardware. Thus, memory leakage would be a highly detrimental bug. It is however highly questionable if memory usage would have a high impact in our model. It is an important metric that must be tested. But, unless there are faults found that are so severe that bug tickets or customer complaints are submitted, it can be hard to rationally correlate this to the release quality.

One of the metrics that we found to already be extensively collected is boottime. There are different kinds of boot times. For example after a hard or soft reset. It is also a metric valued highly by Axis. The goal is of course to have no downtime but should it happen then the boot time hopefully keeps it for as short as possible. The metric we use is after a hard reset, on suggestion from Axis.

CPU usage might not be as important as the other two metrics for Axis cameras but there is much data of it and easy to use and test in our model.

As with CPU usage, system ready time is not the most important stat. However, it is a metric that customers prioritise being high. As such it can not be ignored. Furthermore, the case company is tracked the metric, it has relatively good data history and very easy to extract.

Development metrics

Development metrics measure attributes of the development process. These metrics can be computed using information in version control systems and change management systems.

We had a lot of discussions with our Axis supervisors about which metrics would be useful to use within this category. For example, the number of commits might be considered as a development metric. But after the discussions we figured that the number of commits does not really say much about the quality. A metric that our supervisors strongly liked was how the packages change from one point in time to another. The points in time of interest in our case will be explained in the next section.

- Release blockers
- Number of bugs
- Number of showstopper tickets
- Package changes

Release blockers is beforehand perhaps one of the most reliable metrics, in the way that if there are a lot of release blockers then you have on paper that something is not right. This of course requires that the personnel categorizing the tickets do not put a release blocker on every single ticket but only on the ones actually causing harm. Even without our model, the company is not releasing when there are release blockers, clearly showing that this metric is

relevant to include.

Number of bugs is similar to release blockers because it tells that something is not completely right. Logically reasoning, more bugs will result in a worse release, given that it would be released without fixing them. Some bugs can be release blockers, but it is not always the case. However, counting bugs is probably a good way of self checking the situation.

Number of tickets can roughly be translated to number of problems. A ticket does not have to be a bug in the code but could be any kind of problem. However, as with bugs and release blockers, it somehow tells that something is not right if there are a lot of tickets getting in.

The most important metric among development metrics, along with release blockers, was package changes. Packages are software components that are used in the products. Some products use different packages, some use different versions of the same package, it depends on the product. The interesting aspect of package changes is that they have a big area of effect. A change in one package will affect many products and as such also affect the release. Some packages are more important than others. For instance, a package can have many other packages depending or inheriting from it. Thus, changing that one will perhaps require a change in the other ones. Furthermore all package changes have the risk of increasing the risk of something going wrong which can decrease the quality of the release for the moment. Since the opposite is not the case, i.e. if you change in a package it will definitely improve quality, is not the case, one can see it more as a warning flag. In other words, if changing certain packages then it is likely that something will go wrong.

Package-Change Metrics

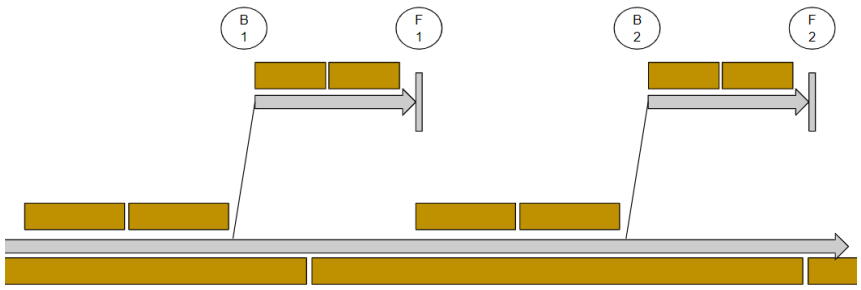

Figure 4.4: An illustration of the branch and test structure. The arrow-blocks are the main branch and two release branches. The darker blocks are various testing phases, where different tests are run. The B-circles represent the point in time when the release branch is born, and the F-circles is when they are released. Note that the F-circles only exists on the release branch and not on the main.

When looking at package changes, there are various time-frames that might be of interest, as shown in Figure 4.4. These are intervals that can be used when viewing the data where B stands for branch point and F for final release:

- B1-B2, *bb*
- F1-F2, *ff*

- F1-B2, *fb*
- B1-F1, *bf*

As mentioned in Section 3.5.2, fixes are added to the release branch and main branch but new features are only added to main.

bb means that looking at the changes from B1 to B2 would show the changes to the main branch, and these changes include both features and fixes. This would mostly indicate the evolution of the software over time, as it includes all changes. It might however not reflect the quality of the release, as the actual release F2 is some time after the branch point B2.

Looking at *ff* would show the changes that were made between two releases. This would directly include the changes that take place on the main branch from F1 to B2, as well as the fixes on the release branch from B2 to F2. In addition it would also indirectly include both the main branch and the release branch from B1 to F1, as B1 is the point where the two programs split up. This means that looking at F1 to F2 would include looking back in time to some extent. Thus, this is the case that should show the biggest amount of change between versions.

As for *fb*, it is of little interest. This is basically the same interval as F1 to F2, except that it is only a part of it. It suffers from the same effect, in that it goes back to B1 in order to find an intersection point for both programs.

As for the *bf* case, we would not be looking at the main branch at all. This would simply be looking at the release branch and thus the fixes that are introduced there. This should be the one that show the least amount of change, but the changes that it does show should be the most volatile for release quality.

When deciding how to store package changes as a result, we had several ideas. We did not only want to count the number of changes, but also how big they were. First of all, there are three ways a package can change between releases. It can get a new version, it can be added as a new package, or it can be completely removed. This gave ground for 3 version comparison methods:

- Added: Only count when a new package is added.
- Removed: only count when a package is no longer included.
- Simple: Count any change.

The tricky part was how big the change was. All versions were not in the same format which gave us trouble because we could not operate on the version numbers the same way for all packages. However, there was a clear majority of one format, which was x.x.x, for instance 3.2.1. This is Axis standard where the first number counts as a major change, the second number a medium change and the third number is a minor change. With this information we created some additional comparison methods:

- Basic: Just count the greatest number to be changed, and multiply the change difference with a respective coefficient.

- Complex: Count the difference in each position and multiply each with a respective coefficient.
- Pos1-3: Only count when changes are made to the respective position.

As for the coefficients, we ended up simply using 3,2 and 1. This was a result of apparent inconsistencies in when, how and why the versioning were changed.

As we completed our first official correlation testing, there was little correlation for any of the time series that were based on all the packages viewed together. None of these time series gave us a correlation value over *0.5*. So it was decided that we should look into individual packages, something that was easy to do as as we had already written the code to do so. We ran the correlations between each individual package and Q_{out} and singled out the time series that gave a correlation greater or equal to the critical value. The critical value is based on the number of data points and a standard significance level of 0.05. This gave us a subset of packages that, in one or more of our comparison methods, gave a significant correlation. What we would like to extract from this subset is which comparison methods are the most useful and what packages those are useful for.

We then analysed the composition of these time series. As we were searching for packages with strong correlation, either positive or negative, we separated the packages into groups based on the correlation orientation. Keeping positive and negative correlation separate, we found that the comparison method *simple compare* was dominant. This was true for both the positive and the negative groups as seen in Figure 4.5. We can also see that there is almost no difference in the prevalence of *ff* and *bb*, whereas *bf* is barely represented.

We then proceeded to look at only the packages that were present as either *simple-compare ff*, or as *simple-compare bb*, in order to see whether *ff* or *bb* is the more profitable comparison method. This meant that we ran our package comparison program on only the packages that were present in the subset as one of the above. We wanted to use the same comparison method for both groups, in order to be concise. The results showed that for the negative group, both *ff* and *bb* showed the same correlation down to two decimals. In the positive group however, the *ff* comparison had a correlation that was about 12% greater than that of *bb*.

Finally, we went back to the original subset and singled out the packages present in *simple-compare ff* and used those to make the final subsets of packages: those with positive correlation and those with negative correlation. When looking at these two subsets with the *simple-compare ff*, both showed a great correlation factor of approximately *0.82*.

Internal data

When looking at some of the Q_{in} such as the tests being run in-house, with results such as boot time, we ran into the problem of how to define and divide the data. Many of the people involved, ourselves, supervisors and personnel at the case company, were in agreement that it was change in the values that would be important, rather than just the value itself. As such we needed to transform the data to a format that told us more than just one value for each day. For many of these data, there were more than 20 results for each day. As such we

Positive		Negative	
Number of time-series	297	Number of time-series	1057
Number of packages	97	Number of packages	302
added	0	added	121
basic	58	basic	148
complex	49	complex	145
pos1	30	pos1	52
pos2	66	pos2	129
pos3	0	pos3	2
removed	0	removed	173
simple	**94**	**simple**	**287**
bb	**127**	**bb**	**456**
bf	42	bf	160
ff	**128**	**ff**	**441**

Figure 4.5: Initial results of looking at packages individually. This data represent the time series that cleared the critical correlation value, and is a breakdown of their composition.

computed the mean and variance for each day. The variance not only tells us the unity of the values of the day in question, but looking at the changes of variance can give an indication of that stability over time. As such, we treated mean and variance with equal importance and computed the following for both of them.

- Average of the value over the last x days
- Average of the last x days compared to the last y days
- The nth derivative

The reasoning of all three is simple, that a single day of good or bad values tells us less than a week, or a month. This means that it makes more sense to look a the averages, rather than the individual days. As such, we compute the average value of the last x days, where x is anything in the interval 0 to 30. We chose 30 as an arbitrary endpoint, since we want to look at the current version of the program.

The second statistic is a continuation of the first, where we try to find the change, by comparing the more recent statistics to the slightly older ones.

The n:th derivative is a bit of exploratory testing. It is derived from the fact that we are interested in the change, and also that it is sometimes used in signal processing. By taking a higher derivative, we consume more values into a single one. So our basic idea is that it is similar in nature to average in that way, but we are unsure of what it actually says. In the end we could not draw any conclusions from this which made us just not use any of it.

As we started to look over the values over several days, we came across the problem that some days lacked test values. So we tried some methods to fill in the gaps.

- Inserting the last known value
- Inserting a halfway value between the last value and the upcoming value
- A gradual connection of the values before and after.

All three ways have various arguments in their favor, but in terms of statistics they all have one fatal flaw. because they all base their value on the once around them, those values gets over-represented when we calculate things such as averages. We then looked into the possibility of simply removing those empty days from the data.

After analysing what dates and weekdays were without data, we found the following:

- Most of the missing days were weekends, but not all weekend days were missing.
- There were several other weekdays missing, mostly but not completely comprised of red days.

With this information in hand, we decided a strategy for dealing with these. We decided on using what we felt was the logic of the scenario: "If the model was used on that day, what would be of interest?". As such we would be interested in all the data that were available, so the weekend days with data were kept. But other than that, weekends are non-working days and as such, little changes in the software would be made, so we would not count weekend days without values. We are only interested in the data that is factual and real, so we will ignore any attempt to fill in the gaps in the data. We are only interested in recent data. So in terms of relative dates, we will count all the week days, even if they are without value. But when counting average, we will only count days with values.

Once we had come this far, there were still quite a number of strange days that threw our different values of. Upon closer inspection of these, it turned out that in some cases we had outliers. One or two values that threw the mean of course on some certain days. So then we had to address the question of whether to dispose of the outliers or not. Most of the outliers that we had noticed were either 0% or 10000% of the surrounding values. And the problem is that we do not know what causes these outliers. It could be a fault in the product that causes a fatal exception, if so, we are interested in those values. It could also be a fault in the test rather the product being tested, if so, we might want to ignore it. Because of this, we decided to test both cases.

This ended up giving us a near infinite number of time-series, calculated in different ways, that we needed to weed out. Once we had written our correlation program we ran all these series against Q_{out}, and analysed the result. We did this by separating every series whose correlation was greater than the critical value. The series were separated by positive and negative correlations as well as their respective natures, much like how we did with the packages in Section 4.2.4. The difference only being that an average was needed to be calculated for

each calculation method. Each original statistic was then scrutinized individually, and a calculation method was chosen for each. The final selection were the same for each statistic, with less than 3 days difference in their averages. This finding was good, as it meant that we could be concise when using the different statistics. What was not as good was the calculation method that our analysis had selected. The calculation that was selected was: "*The average mean of the last 11 days compared to the last 23 days, as calculated 25 days before branch date*". This is not a selection that we can defend logically, even if it was good statistically. At this point we wanted to remove these metrics from our data set. However, as we had derived them correctly, we decided that we would keep them and hope that they would improve any of the good models at the later stages.

Tickets

With tickets, we have a very simple way of counting them, "how many came in since the last release". But this hides much information that we early on came to understand might be important, such as relative numbers when close to branch or release. As such we were interested in processing this data source similarly to how we did with the internal data in Section 4.2.4. At that point, we already had the code for it. We modified the software we had written so that it would work for tickets and the oddities that came with them. One such oddity was that a day without incoming tickets is still a valid data point. This is unlike how it was with internal data, where we either had test results or not. However with the tickets we had to keep every single day when calculating, even if tickets on weekends were very few in number.

The outcome of this turned out somewhat disappointing, as the time series that passed the critical value test were few in number and had no clear structure. We spent a bit of time trying to see if there were anything in the data that was dominant and we could use, but in the end none of these series made the final selection. Thus all the ticket Q_{in} metrics we have are based on indiscriminately counting the tickets in the time frame since the previous release.

The demise of metrics

We ended up having a large amount metrics after the study and interviews, i.e. metrics that could be used as input to the model. Decisions had to be made about which of them to use. A draft of the metrics we had ranked as described in Section 4.1.2 can be found in the appendix. There were some metrics that even though they were included late, the supervisors from Axis deemed important and wanted us to include. The metrics were *package changes* and *release blockers* and were easiest to make a decision about. For all of the other metrics found we had to make our own decision. There are too many to go through each one of them in detail so we will try to summarize. Also in the appendix can be found another matrix that shows the final version of which metrics were used and which ones were not. In the second matrix there is one difference to the colouring compared to Section 4.1.2. Only the ones actually used in the final input are now green. Those that were green before but that we did not use are now yellow. Those that were yellow are red, and the red ones are still red.

The final metrics used are:

- Package changes

- Release blockers
- Boottime
- CPU usage
- Memory available
- System ready time
- Number of bugs
- Number of showstopper tickets

It seems that code metrics will not be a useful entity to our model. We talked to several of the people involved in the RD departments at the case company, and we got to the understanding that it would be very hard to gather the data needed to insert metrics into our model. Furthermore, the structure of the software is that of modules that are coupled together. Which means that it is not the complexity or fault proneness of the code in isolation that causes problems, but rather when modules are connected. As such it seems that most of the code metrics will neither be easily collected nor robust, as defined by Peng and Wallace [24], on what a good metric is.

When we looked deeper into the API of the ticket system and had written some programs to extract certain information, we came to the realization that every ticket had but a single timestamp. This was the timestamp of when the ticket was created, meaning that this was the only piece of history we could find for the tickets. We can only see the final values for the tickets and when they were created. This finding put a stop for a great amount of our potential metrics. It meant that it was impossible to see what the status was for the tickets at the time of release. The fact that we only have one timestamp per ticket also means that we cannot extract when the ticket was solved or accepted, putting a stop to all the metrics that require a status of the ticket at release such as:

- Service request time
- Mean time to repair
- Number of bug fixes compared to history
- Open defects
- Critical problems prior to release

When we started to work on the gathering of the test data it quickly became apparent that there was not a lot of history available. The data that was possible to extract only represented about a year in real time, representing only about 4 releases. We quickly realised that there would not be enough data to use for our model and stopped working on trying to extract information from that system. This finding set stop to the following metrics:

- Test failures

- Number of tests
- Number of passing tests
- Defect density (defects/size of the software)
- Reproducibility
- QA test data

It should be noted that the internal data is a separate system from the QA test system, so this finding did not impact metrics such as "CPU Usage" and "Boot time".

Now there are some metrics that we marked as green in the start, which means we thought of them as relevant and useful, but in the end did not use and are not included in the bullet lists above. They are:

- Flash usage

  Flash usage is an internal metric like boottime, CPU usage, memory available and system ready time. It can be argued that none of these metrics affect the release. However, with the flash usage case, it is even lesser important for a product and thus leading to the reasoning that it simply does not seem relevant for how good a release is. For memory available, to compare with something, if the memory available for a product is bad then there is a risk it will break if shipped, the code is probably written bad and it contributes towards a bad release. Of course no single product will make a release bad. Looking at flash usage the same way, it does not contribute as much.

- Response time

  Because of the same reasons as flash usage above, we decided to not include response time either.

- Master prio

  Master prio is a category tag that tickets can get. Same as *release blockers* for instance. The reason for not including the number of master prios is mainly because we already have release blockers. The two are in principle the same thing, only release blockers is more crucial. A release blocker will do what its name suggests. A master prio will not block a release but it is prioritized on the master branch. Often master prios become release blockers when branching, but a release blocker can also be downgraded to a master prio if it shows that a release still can be made without fixing the issue.

- Defect find rate

  This would be defects per time-unit, or possibly how the current defect per time-unit is compared to the usual one. In the structure of our case company that corresponds to bug tickets. As such it was set aside along with some similar ticket related things in the procedures described in Section 4.2.4

- Number of commits

  The number of commits is an interesting metric, to see if a lot of commits affect release quality. It could be seen as a size metric. We wanted to include this one as input but unfortunately it was simply too hard to extract. The company uses gerrit to commit code but we could not process the information and count commits in a smooth way.

- Number of firmware images

  A firmware image is all the software that goes into the product as a file. A product can not have more than one of these. It essentially contribute the same as the number of products does since almost every product have a different firmware image, resulting in almost the same number. Furthermore, the number of products is used in Q_{out} which is why we decided to not include it

4.2.5 Constructing models

When looking at the different models we could produce and which was best we used two main methods.

- PCA(Principal component analysis)
- Stability Comparisons

PCA was used to check for complications between the Q_{in} metrics, as explained in-depth in Section 3.2.3. PCA is able to group metrics together to create artificial metrics, based on variation similarities of the metrics. We used this to combat multicollinearity. The stability comparisons is something we made ourselves, and is just as the name implies, a way to check the stability of the model. The procedure of this is as follows.

Once the metrics are selected, remove one of the data points and train the model on the rest. This will produce a regression with unique coefficients for each metric as well as a unique remainder. This regression is then used to predict the removed data point. Then it is possible to compare the predicted value and the real value and thus get a quantifiable error. This represents one set of training results, where the product is an error, a remainder and coefficients. It is possible to construct one of these sets for every data point.

By looking at these sets, it is then possible to see both how big the errors are and how volatile the coefficients are. By volatility we refer to the idea that these sets preferably have the same values for the coefficients, so that they can be merged into one single model with one set of coefficients. If the coefficients are too varied it would mean that the model is too dependant on specific data points and that it can not be trusted. Putting these together, we are looking for a set of Q_{in} that gives a stable model, that produces a low error margin and shows a low level of compability problems in PCA. We had an assumption of what Q_{in} combinations would be useful for the model, but we wanted hard data. So we wrote a program that would produce the stability comparison results for every set of of the Q_{in}s that had cleared the previous tests of correlation with Q_{out}.

When finally looking at all the models produced, we mainly looked at the 3 properties *average coefficient difference*, *average error* and *max error*. The average coefficient difference is the average size difference between the highest and lowest occurrence of each coefficient and remainder. In addition to these three, we also took number of metrics and source of metrics into account when scrutinizing the models.

The general structure of the results were that the best models had few metrics, and none of the internal data metrics. The best model was one that contained only the metric "positive package changes", so objectively we had to choose that one. There was also one model that used all 4 of the non-internal data metrics, that stood out as it was much better that any other 3+ metrics model. There was only a single property that was bad on it, max error, but when compared to it's average error it was clear that it was a special case for one of the data-points and the rest was quite good. The third model we chose was a 3 metric one that was only slightly worse than the first one, despite having so many metrics.

Since the idea is that Axis will continue using the model/models, and we have such few data points, the final models will be trained on all data points available. Thus the validation data will not be exactly extracted with the final models.

Chapter 5
Results

5.1 Interviews - Coefficients and Old Releases

There were a total of 4 separate occasions for these interviews. The first two interviews were with members of Product Management. One of these had been directly involved with the release process for the last 3 years, and had additionally been in charge of several of them. This was the most giving discussion, as this person had a lot of experience and could remember the most distinct releases. The releases that this person remembered as especially troublesome were releases 11, 13, 17 and 19. They noted especially release 19 as strange. It had seemed like the best release so far, but when released there was a new hidden error that was found. For release 13, the problem was not that it was especially bad, but rather that it was planned to be an LTS. Once it was released some problems were found, this caused the creation of the LTS to be pushed onto the next release.

The other person in the interview was one of our two supervisors, and had been directly been involved in Product Management for less than two years. They also mentioned the recent release 19 as being a hidden bad release. Otherwise they could remember few differences between the releases they had wored on. Another point made was that unique products tended to generate more problems than those that were similar to other products. This was also the first time we found out that the only release to be given support is the current one and LTS versions.

The Third interview was a double interview with two members of the QA department. The first of these members was the leader of the department. The second interviewee was one of their direct subordinates. The results regarding previous releases quality were sparse. The only distinct release quality information they could provide was a repetition of the release 19 information. Furthermore it was made clear that the QA apartment had slightly different

priorities from the rest. Because they only work to increase quality, they were in fact positive to release delays, as this allowed them more time to run tests. It was also confirmed that problems that are found late tended to make a greater impact than the rest.

The fourth interview was with one of the product specialists. In this interview there were results for releases 13,17 and 19. For releases 13 and 19, there was no new information. For release 17 there had been an update in one of the supporting soft wares, causing some products to be pulled irrespective of quality. In this interview it was also confirmed that the only releases to be given support were the current one and the LTS versions.

The final priority rating ended up as shown in Figure 5.1. These numbers were then used directly as coefficients for the equation described at the start Section 4.2.3.

Number of products	2.74	
Pulled products	39.15	
Delays	23.59	
Bug-tickets	6.68	
Priority	16.03	7.35
		2.02
		0.72
		5.93
Severity	11.81	8.16
		2.54
		0.69
		0.42

Figure 5.1: Final Q_{out} coefficients. These are used in a simple $C_1x_1 + C_2x_2 + C_3x_3$ equation to produce Q_{out}. Priority and Severity are characteristics of tickets and are represented both as a whole and as their sub-categories. For priority the subcategories are High, Medium, Low and Incoming. For severity the subcategories are Showstopper, Serious, Medium and Small.

5.2 Models

In the end, we had 8 input metrics. By doing combinations, i.e. using the different metrics together in all possible ways, we could make 255 different models. The models are in fact just equations, based on linear regression, that use a set of metrics. The values for the metrics used corresponds to the models Q_{in} and the prediction itself is the Q_{out}. In this section we will present the three models that we chose as well as their prediction results, errors and stability.

In the equations for all of the models we define the following:
x_1 = number of package changes with positive correlation to Q_{out}
x_2 = number of showstopper tickets
x_3 = number of package changes with negative correlation to Q_{out}
x_4 = number of bug tickets

5.2.1 Model 1

- Metrics: Package changes (positive correlation)
- Equation: $5,77x_1 + 493$
- Prediction: 614

Model 1 uses only one metric, package changes with positive correlation to Q_{out}. This is a measurement of how many changes there has been in all packages between one release and another. This was measured by iterating through a file for each release, containing name and version for packages used in that release, and then counting changes. For instance, if there exists a package for release 1 but the same package does not exist in release 2, that is a package change. In the same way if it does not exist in release 1 but is added in release 2, it is also a change. The last thing we count as a change is if it exists in both releases, but has a different version in each.

The equation for all models are auto generated from the available data. In this case the coefficient is very high. This means that each package change has a high negative impact on the quality, since our Q_{out} has an inverted scale, i.e. the lower number the better quality as described in Section 4.2.3.

The prediction was what the model predicted on the last release or data point that we had available, after being trained on all the rest. The number of these predictions can not unfortunately give a definite answer on if the quality is good or bad. Instead they can be compared to each other. This means that if the prediction is 614 for the latest release and it predicts lower than that on the next one, then the next is better, else it is worse. But as the validation results show 5.3, the models predictions are not 100% accurate.

5.2.2 Model 2

- Metrics: Package changes (positive correlation), showstopper tickets, package changes (negative correlation)
- Equation: $3,22x_1 + 0,57x_2 - 1,82x_3 + 548$
- Prediction: 622

Model 2 uses three metrics. The first metric, package changes is the same one as in the first model. The other two are new. Tickets can have different tags to easier keep track of them. One of those tags is a severity tag. There are four levels of severity, where showstopper is the most severe one. This metric counts the amount of tickets that have the showstopper severity tag in between two releases.
The second metric is package changes with negative correlation to Q_{out}. This works the same as package changes with positive correlation. Even if the correlation is negative, if it is highly negative, it means that it has an impact on Q_{out}.

The equation shows that package changes with positive correlation still has the highest negative impact. An interesting thing is that package changes with negative correlation has a negative coefficient. This means that the higher the value for that metric is, the better quality will be predicted by the model.

The prediction on the last release was 622, which is higher than what model 1 predicted. This means that according to model 2 the release will be a little bit worse than what model 1 predicted. They did not differ much, only 8, and the models are not 100% accurate. That is a reason we choose to show three models. We believe it gives a better feedback to the case company.

5.2.3 Model 3

- Metrics: Package changes (positive correlation), showstopper tickets, package changes (negative correlation), bug tickets
- Equation: $2,38x_1 + 0,73x_2 - 1,78x_3 - 3,63x_4 + 598$
- Prediction: -561

Model 3 uses four metrics, which is the most of the three models. In addition to the other three metrics which are used in the other two models as well, model 3 includes bug tickets. Bug is a tag that tickets can have to easier identify what kind of ticket it is and to keep track. Thus, this metric is a count of how many tickets are categorized as bugs between two releases.

The equation of model 3 keeps lowering the coefficient of package changes with positive correlation. The two coefficients for showstopper tickets and package changes with negative correlation are approximately the same. For bug tickets it is $-3,63$ which is relatively high negative compared to the others.

The prediction for model 3 was -561. This suggests that according to this model, the release should be extremely much better than the others.

5.3 Validation results

Table 5.1 shows an overview of the results after the validation process. Average error is the average error from the corresponding tables below, where the error is the difference between the predicted Q_{out} by the model and the real Q_{out}. Average coefficient difference is how much the coefficients differ in every equation you get from the training changes.

The Tables 5.2a, 5.2b and 5.2c show how well each model predicted earlier releases. From the tables we can see that the error interval for the first model is (8, 147), for the second model it is (17, 142) and for the third model the error interval is (2, 532). The predictions for all of the models are roughly in the same interval, approximately between 500 and 800. One model that stands out a little bit more than the other two is model 3. It has the lowest minimum error of 2 but also the highest maximum error of 532, of the three models. Furthermore, it is

model	metrics used	average error	average coefficient difference
1	Package changes (pos)	64	26
2	Package changes (pos) Showstopper tickets Package changes (neg)	64	29
3	Package changes (pos) Showstopper tickets, Bug tickets Package changes (neg)	90	30

Table 5.1: Overview of the three best metric combinations.

the only model that has predicted a negative value for one of the releases.

After presenting and discussing the results, the relevant personnel to our project agreed that the predictions looked good and were satisfied with the error margin. Since there is no previous work or numbers to relate to and compare, it was a little hard to make any conclusions. We presented all the results that we had worked with, training data as well as current release. This allowed them to compare different releases and although the absolute numbers of our computed quality are vague and means little by themselves, when put together with other releases they could make sense of them. According to the validating data and their own experience with the releases it seemed good. The presenting of the results can be seen as informal interviews which were most free discussion. However, we asked them two important questions directly, which were if the results look good and if it is something they can consider keeping and work further on. On both questions the answer was yes.

The metrics that were used is something that we have had discussions about during the whole process of constructing Q_{in} and Q_{out}. Thus, they were satisfied with them and positive to the metrics having some theoretical weight as well as logical. Package changes in Q_{in} and pulled products used in Q_{out} were two metrics that should stand out a little more than the other according to the personnel. This was also the case in the models, except for one case where bug tickets were included.

	Model 1	
Release	Prediction	Error
r11	557	48
r12	525	19
r13	558	97
r14	565	147
r15	554	24
r16	536	152
r17	594	29
r18	802	58
r19	781	41
r20	560	77
r21	650	8

(a) Model 1 validation results

	Model 2	
Release	Prediction	Error
r11	546	59
r12	523	17
r13	581	74
r14	570	142
r15	549	19
r16	509	125
r17	658	35
r18	800	56
r19	799	59
r20	548	65
r21	592	50

(b) Model 2 validation results

	Model 3	
Release	Prediction	Error
r11	631	26
r12	530	24
r13	611	44
r14	622	91
r15	400	130
r16	-148	532
r17	656	33
r18	746	2
r19	748	8
r20	576	93
r21	648	6

(c) Model 3 validation results

Chapter 6
Discussion

6.1 Models

We define a model as a composition of the metrics used and the equation it got. As such we will not only discuss the result values but also dwell into the metrics. A difficult thing about the results is that we can not really relate or compare them to anything other than each other in order to get some better understanding.

6.1.1 Metrics used

We got three models in the end, each with a different number of metrics. They did not differ much. Only one noticeable thing was that the average error of the third model which uses most metrics, exceeds the other two models with approximately 30% and 50% respectively. Other than that, the average coefficient difference is roughly the same and the prediction interval as well. In each of the models, we can see that package changes with a positive correlation is included. For the other two, showstopper tickets as well as package changes with negative correlation is included. In the last one bug tickets is included, with a negative correlation, as well.

Beforehand, package changes is something that we assumed would have some significance. The reasoning behind package changes made sense. If no code is touched then obviously the next release will not change much either. It is not known beforehand if changes in a package will have negative or positive effects, but what is known is that it will have some impact, which makes it a good metric to use and not really a surprise it is included in all the models. We have two types of package changes. Those that have a high positive correlation with Q_{out} and those that have a high negative correlation with Q_{out}. What is important here is that both types have an impact on Q_{out}, only in opposite directions. Showstopper tickets and bug tickets are also metrics that are reasonable to be included. A first thought is that

showstopper tickets and bug tickets directly affects the quality negatively, which is the case for showstopper tickets, but not for bug tickets, as can be seen in the equations for model 2 and model 3. We will discuss this more in Section 6.1.2.

Metrics that are not included in the models are the so called hardware metrics, *memory available*, *boottime*, *CPU usage* and *system ready time*. This was not too surprising. First of all they showed low correlation. Furthermore, individually these metrics do not reasonably have a big impact on the quality of a release. If they all show good values, a product can be said to be good, but the release itself does not necessarily have to be better because of that. Furthermore, when we are looking at the values we are lumping all products together. It is very possible that the merger of metrics from different products is impairing these metrics. If a deeper analysis of these metrics were to be done, where products or product groups are observed individually, then the result might be very different.

6.1.2 Model equations

The equations produced come directly from the linear regression. What is interesting is how the different coefficients varies. For instance, positive package changes becomes less and less significant as more metrics are included. Showstopper tickets and negative package changes on the other hand do not change that much between the models they are included in. One might assume all the coefficients to be positive except for negative package changes. However, as mentioned above, this is not the case. Bugs for instance have a negative coefficient, indicating that much like a rainbow after rain, a good release comes after a bad one. The reason for this is debatable, but one hypothesis is that when there are many bugs, the developers are more careful with what they merge into the project, resulting in a better release.

A kind of verification is that negative package changes have a negative coefficient for both models it is included in. This is expected since when correlating the metrics individually against Q_{out}, negative package changes got a negative correlation, hence the name. This means that when negative package changes is increasing, Q_{out} is decreasing, and vice versa.

There is not much to say anything about the intercept value. Why they are so high probably is because of how the Q_{out} is calculated. For the model's predictions to be able to get up to the Q_{out} values a high intercept is needed. The real Q_{out} values trained on are approximately in between 300 to 700 with an average of 600. The intercept of the equations differs with approximately 100 from the first model to the third one. Why it is increasing perhaps depends on the number of metrics. In the first model there is only one but with the highest coefficient of all. In the third model there are four but with somewhat lower coefficients.

6.1.3 Prediction results

The prediction results were a little bit mixed. Two of the models predicted slightly above their averages whereas model 3 predicted -561. This might seem strange but when analysing the inputs it becomes clear. As explained in Section 6.1.2, bug tickets has a high negative coefficient. This combined with approximately 350 bugs tickets between the two releases of

course gives a very negative prediction. As for the positive coefficient metrics, positive package changes had a value of 9 and showstopper tickets had 41 which cannot really measure with the number of bug tickets. This makes model 3 special in the case that if there are many bug tickets then it probably will predict a negative value, which is something that the users of the main script should be aware of.

The validation results at a first glance look somewhat similar. The only model that stands out is the third model. It is the only one with a negative prediction, which is the lowest of them all. Overall, the third model has the lowest average prediction of 547, whereas model 1 and model 2 averages 607 and 606 respectively. The reason for model 3 predicting low, points to it having two negative metric coefficients in package changes (negative) and bug tickets. Especially bug tickets, with the highest (negative) coefficient seems to contribute much to this. The only difference between model 2 and 3 is bug tickets. Why bug tickets have this effect depends on the model produced from the linear regression. But something that should be considered is that bug tickets is used both as Q_{in} and in the Q_{out}. Another thing is that for the releases where model 3 predicts much lower than the other two models, it can simply be so that there were a lot of bugs which the other models can't include. Taking it one step further, a reason to many bugs could be that a firmware was released, and only afterwards it was found out to have bugs. These bug reports will then count as Q_{in} for the next release prediction but as Q_{out} for the released firmware. The predictions follow roughly the same patterns. For r18 and r19 they all predict higher and for the rest it is more concise. Whether the predictions are good or bad is hard to tell. All we know is that the lower they are, the better the quality of the release. By only judging the validation results, 400-550 would seem good.

6.1.4 Prediction errors

We saw that the errors are widely spread. That is not a good sign regarding the models. It means that they are inconsistent or unreliable when it comes to accuracy. However, Table 5.1 shows us that for model 1 and model 2 the average error is 64. The same two models on average predicted 607 and 606. This approximates to a 10% error margin which we believe is good enough. For model 3 it is a little bigger. By categorizing the error in intervals of size 10, i.e. 0-9, 10-19, 20-29 up to 140-149 and 150 or above, we get that the interval with most occurrences was 50-59 with 5. The second most occurring was a tie between 0-9 and 20-29 both with 4 occurrences. This tells us that the predictions were not that far off which is good. The reason for using the interval size 10 is because of how the values look and it enough comfortable and manageable.

6.2 Validation with the case company

One thing that they noticed quickly was some of the predictions from model 3 that deviated from the rest. This is a thing they were concerned about. But they understood the results after explaining why model 3 behaves as it does. This is however something that they will have to be aware of in the future. If a release should have a very higher amount of bug tickets compared to package changes and showstopper tickets, then they should expect the predic-

tion to be very negative from model 3.

Throughout the whole project, the case company has been very keen to see the results. What the models look like and what they predict. After all, what they want is to see is the quality of a firmware release in numbers. However, the whole methodology is also something that they were interested in. We have not only just created three different prediction models, but also showed them how it can be done. With that said, the company can in the future add other metrics they find fitting and create new and maybe better models. Furthermore, it was nice to hear that they came up with ideas how future master's theses could be followed up on our. To sum up, they would start by using our numbers and in the future look more into our work to customize the models.

The personnel that were going to be using our result, were very interested in the Q_{out}. Much like us they have been having problems with quantifying the quality of a release. As they had been part in the AHP surveys, they were versed in how we calculated the Q_{out}. In fact, the final coefficients that were used after the merger of the surveys were very close to the individual answers that they had given. Because of this the Q_{out}s were very similar to how they felt about the releases. The only exception being a recent release that had been problematic. The problem with that release was a bug in an area that was arguably outside the scope of the release work. As such it did not have a large influence on our model, but of course it's large impact on the work in the case company. So they were interested in manually correcting certain releases.

6.3 Newer findings concerning packages

As we were writing the program that implemented the models we had produced, we came across some slightly disturbing findings. We found that our package metrics were less prevalent then previously. Upon some investigation, we found that not only were there very little change in the version numbers of our selected packages, but also a few of them were simply not included anymore. This raises some interesting questions on the usage of specific packages as metrics. Some of our most reasonable hypothesis are:

- The packages were concerning functionality that gave rise to problems, but those functionalities are now implemented in full. Thus the packages are now mostly stable.
- The packages are only used in discontinued products, and are thus currently only included in the LTS's.
- The packages have been renamed or the entire code areas that the packages were part of have been entirely rewritten.

Independant of the actual reason, it is clear that using these metrics will require continued work. Analysing what sort of packages it was that gave positive and negative correlation respectably. What was it about these packages that made them correspond to our Q_{out}? Are there any factors that are common amongst them? These are some of the potential answers that we are thinking of:

- Hub-packages that connect functionalities?
- Hugely complicated niche functionality that might not be worth the trouble?
- Important core functionality that everything depends on?
- Clusters of low code standards?
- Seldom used packages that are barely tested?

Whatever it might be, if there is any strong trend shown amongst these packages it should be possible to use that knowledge to better the development of the software.

6.4 Changing systems and loosing data

6.4.1 Quantifying quality in a large and undefined area

This area of work is very large and we have been forced to make many assumptions and simplifications. What differs our work from what we found in literature is the level that we look at. A lot of papers in our literature study focused on the code at different levels. We found some that only gathered softer metrics, i.e. metrics that have looser definitions or that can be open to subjective bias. We also found some that only just gathered and presented the metrics. What is different for us is that we are working at a higher level in the company. We are interested in all aspects that might conceivably have an impact on the release quality. With that broad spectrum comes the important question of what quality is. When looking just at code level, it is a bit easier as there are quality metrics such as actual bugs found as well as a more concise history, as everything is digital. However, with a scope as wide and a level so high as ours we cannot take code quality as our quality indicator. Although bad code metrics and quality might work on a lower level case study. When looking at company level, it is not useful for giving quality to a release. When looking at quality at our level we almost have to look at more subjective quality, both from the customers' and the personnel's sides.

What would have been nice is if customers filled in a small questionnaire each time they upgraded their software, about possible problems they've experienced with their current version. And possibly do the same for personnel at the company, because as is, no one could really remember what they experienced during the earlier releases. This would give more solid ground to the Q_{out} and thus the entirety of the models. As it is we had to resort to what was available, and we are not completely happy with what the results of that. We had a slight insight into the customer response in that we had access to the CST-tickets, but we do not know what might have been cleaned away in the CST-ticket procedures. As for the other Q_{out} components, they are more related to the end of the release rather than what is delivered. Thus they are an insight into the status inside the company during the release, which we reason also is a part of the quality of a release.

6.4.2 Low amount of data points

A well known approach to improve models such as this and for machine learning models is to increase the data points which can be trained on. We unfortunately had to remove almost half of the data points we intended to use from the beginning because there was no data so far back in time. We ended up with 11 data points which is very little. As our model is predicting the quality of a firmware release, it is understandable that the case company simply cannot create new data points now and then, but only after each new release. What is important here is to keep all data to be able to train on as many points as possible in the future. Another problem we encountered was that we had to exclude metrics because of this. Metrics that perhaps could have made the model better.

6.5 Threats to validity

We have attempted to address threats to validity in the corresponding parts of the report, but here is a more general discussion on the subject.

Some of the information gathered manually comes from manually composed reports over releases. Depending on who wrote them, and at what time, both formats and standards differ. We have consulted the people that are currently doing the corresponding work to determine how to interpret various uncertainties, but this is still a possible source of errors.

We have been collecting information from documents and been storing them in spreadsheets for analysis. It is possible that said information have been slightly altered in this step. We have been thoroughly double checking information so that this sort of problem shall be eliminated, but the possibility exists. One example of this is the "." vs "," in numbers problem, where python and the American numbering systems use dots, but the Swedish system uses the comma. And whenever we initialize a new document it started with Swedish by default.

The problem with using the APIs, mostly trouble is validating the data. First of all, the API and the web interface do not work the same regarding dates. We used the API to get the tickets, sorted by dates. For instance, if we wrote "2019-10-25..2019-11-25", it would give us all the tickets from 2019-10-25 to 2019-10-24. So the last day is not included. In the web interface this was not the case, there the last day was included. This made it cumbersome when making sure that the results from the API were correct, which we had to do manually. The biggest problem however, was that the API could not take large queries. To solve this we had to make many more smaller queries and then check that each one of these were correct.

For good and bad, one of the authors have been doing work at the case company previously. This could incur that there is some unconscious bias, such that certain aspects being ignored because of previous experience. This can be good if the context of the experience and our thesis is similar, as it could speed up the work. However, if the context is different then important details could be swept away before they have been noticed. Luckily we are two authors and we hope that the insistent questioning and inquiring of the other half have minimized this as a problem.

In the research and gathering of data in this thesis we wrote scripts in python. As in every project involving programming, there might be bugs in the code. We did not use any testing frameworks to test our code. Instead, we simply ran the scripts several times with different inputs and checked it with print outs and visually against the systems from where the data came. One thing that could possibly increase the chance of bugs, even if we do not think it did, is that we had to write the code in such a way that was fitting for the case company and their code standard and templates which we were not used to. In the end, the scripts were not that big and most of them focused only on one thing. As such we do not believe there were any bugs.

As this is a case study for the top layer of the release process, which is a single department, there is a definite possibility of bias. The case company was not the best prepared for this master thesis. The idea was good but many things were not clear from the start. Several definitions were missing, for instance, what quality actually is and what data is available. Throughout the project we got a lot of free hands so to speak. In other words, there were not many involved from the case company at the start, regarding both programming and determining metrics. As the project neared its end, the personnel became more and more involved in choosing models that we had developed and reviewing our code.

Throughout the master thesis we used proven methods and theories to aid us on the way, such as AHP, PCA and in the correlations. These methods often include various threshold values. We have not been scrutinizing these values and have used the base values that we have found. In case there are several values we have used what have been referred to as the standard value, see critical value for the Pearson correlation for an example. Using other values than the ones we used might give different results.

We have mentioned that we had to make definitions of our own. For instance when using the metric package changes. What a package change actually is, perhaps depends on who you ask. A comment for instance is a change in the file but it does not affect the code or the firmware at all. As explained in Section 4.2.4 and 4.2.4 we tested and defined it in our way which of course is a threat to validity. These definitions were made solely on our own knowledge and intuition, as well as discussions with our supervisors from the case company, and not on any literature, theory or other practice. This threat applies not only to package changes but all the definitions we had to make up ourselves. That includes for example our definition of quality (Q_{out}) as well. As we have been stating in the rest of the report, our distinct results are entirely dependant on Q_{out}. And there is a distinct possibility that with another Q_{out} the results would be different, positively or negatively. During this thesis We have done our utmost to be as objective and personally unbiased as possible. According to the project leaders, the procedure and results were possitive, as such we can only hope that the result is sound.

Most of our data is gathered from objective systems that are largely trustworthy, but others are subjective opinions based on peoples faulty memories. It became apparent that using peoples distinct memories was not a fruit-full endeavour. So we only used it sparingly, only as a source of inspiration and understanding. We did have to use the subjective opinions in, for example, the Q_{out} coefficients. We hope that opinions such as these are more trustworthy,

as these are subjects that are continuously reinforced during their work.

Chapter 7
Conclusions and Future Work

Through our work we have tried to answer the question of whether it is possible to predict the quality of a release by creating models which can do just that. The models are auto generated equations based on linear regression, that use a set of metrics as the equations unknown variables. The values of the metrics have been extracted from the case company. On the question of whether this helps the case company, the answer is yes. It becomes an extra input to the decision making about the release, based on facts, instead of just their gut feeling. There have been problems throughout the work but it was still manageable to succeed with our goal to create a prediction model. However, to do this again or continuing on it, the points bellow are of importance.

There needs to be a clear definition of "release quality", which is quantifiable. The work is, in essence, trying to find variables that oscillate in unison with the quality of the releases. Without clear numerical data on the quality of the releases, it is simply not possible to make any good predictions.

More data points are better than few. To make a reliable model, many data points are needed to better train the model. In our case a data point corresponds to a release. This means that new ones are not added too frequently. However, our case company is smart and plans to increase the release frequency.

Changing systems and practises presents a problem for this sort of work. It makes relating data points to each other difficult, making some metrics inconsistent and unusable, and in some cases making the entire data point unusable. This also seems to be a theme in software companies to keep up to date and change with the time, meaning that they are relatively often changing systems. This could be mitigated if it is possible to merge the results from older and newer systems into a continuous line. This would require good knowledge of both systems, so that it is assured that it is exactly the same metrics that are extracted. Furthermore, the releases that are in the transition period are not to be as trusted, as it takes time for

people to adapt to the new system and the values will be off during this period. Due to problems such as these, it seems to be really difficult to gather compatible continuous data points.

To continue this line of work, all data points after each release should be saved. This is important to improve the models by enlarging the training data. It is also a good idea to try and look forward in time if some metric looks interesting to be added to the model. This requires that there is available data for it which means the data should be extracted some time before to be able to train on it. Regarding metrics it is important to chose those that have continuous data. If a metric is stopped being used, or it has no data, then it should be removed, else the model will be trained falsely.

The supervisors at the case company, which were responsible for the thesis, explained they had ideas of future theses based on this one. For instance, analyse more in detail how package changes affect the release in different ways. Some positive and some negative. It should be possible to analyse what sort of packages that affect release quality when changed. This is outside the scope of our thesis work, but we saw that the potential was there.

This thesis is something that could surely be done with machine learning. We created our model by a simple multiple linear regression. By applying machine learning it might be possible to learn what metrics give more accurate predictions.

References

[1] Firmware. Wikipedia. https://en.wikipedia.org/wiki/firmware.

[2] Gunnar Blom, Jan Enger, Gunnar Englund, Jan Grandell, and Lars Holst. *Sannolikhetsteori och statistikteori med tillämpningar*, pages 358–371. Studentlitteratur AB, 2015.

[3] Gianluigi Caldiera, Victor R Basili and Dieter H. Rombach. The goal question metric approach. *Encyclopedia of software engineering. Institute for Advanced Computer Studies, University Of Maryland*, pages 528–532, 1994.

[4] IEEE Software & Systems Engineering Standards Committee. IEEE standard for a software quality metrics methodology. *IEEE Std 1061-1998*, 1998.

[5] Khaled El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7):630–650, 2001.

[6] Norman E. Fenton and Martin Neil. Software metrics: roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 357–370. ACM, 2000.

[7] Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8):797–814, Aug 2000.

[8] Todd L. Graves, Alan F. Karr, James S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, July 2000.

[9] Douglas M. Hawkins. The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12, 2004.

[10] Wendell D. Jones, John P. Hudepohl, Taghi M. Khoshgoftaar, and Edward B. Allen. Application of a usage profile in software quality models. In *Proceedings of the Third European Conference on Software Maintenance and Reengineering*, pages 148–157, 1999.

[11] Taghi M. Khoshgoftaar and Edward B. Allen. Predicting fault-prone software modules in embedded systems with classification trees. In *Proceedings 4th IEEE International Symposium on High-Assurance Systems Engineering*, pages 105–112, 1999.

[12] Taghi M. Khoshgoftaar, Edward B. Allen, Kalai S. Kalaichelvan, and Nitith Goel. Predictive modeling of software quality for very large telecommunications systems. In *Proceedings of ICC/SUPERCOMM'96-International Conference on Communications*, volume 1, pages 214–219. IEEE, 1996.

[13] Paul L. Li. Predicting field problems using metrics based models: a survey of current research. *Institute for Software Research International, Carnegie Mellon Univeristy*, 2005.

[14] Paul L. Li, Jim Herbsleb, and Mary Shaw. Finding predictors of field defects for open source software systems in commonly available data sources: a case study of openbsd. In *11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 10 pp.–32, Sep. 2005.

[15] Michael R. Lyu. *Handbook of software reliability engineering*. IEEE computer society press CA, 1996.

[16] Audris Mockus, Ping Zhang, and Paul Luo Li. Drivers for customer perceived quality. In *Proceedings. 27th International Conference on Software Engineering (ICSE)*, pages 15–21, May 2005.

[17] Audris Mockus, Ping Zhang, and Paul Luo. Li. Predictors of customer perceived software quality. In *Proceedings. 27th International Conference on Software Engineering (ICSE)*, pages 225–233, May 2005.

[18] John D. Musa, Anthony Iannino, and Kazuhira Okumoto. Software reliability. *Advances in computers*, 30:85–170, 1990.

[19] Nachiappan Nagappan. A software testing and reliability early warning (STREW) metric suite. 2005.

[20] Nachiappan Nagappan, Laurie Williams, Mladen Vouk, and Jason Osborne. Early estimation of software quality using in-process testing metrics: A controlled case study. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, May 2005.

[21] Srinivas Nidhra and Jagruthi Dondeti. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, 2012.

[22] Indira Padayachee, Paula Kotzé, and Alta van Der Merwe. Iso 9126 external systems quality characteristics, sub-characteristics and domain specific criteria for evaluating e-learning systems. *The Southern African Computer Lecturers' Association, University of Pretoria, South Africa*, 2010.

[23] Fabiano Pecorelli. Test-related factors and post-release defects: An empirical study. In *Proceedings. European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, (ESEC/FSE) 2019, pages 1235–1237, 2019.

[24] Wendy W Peng and Dolores R Wallace. Software error analysis. *NIST Special Publication*, 500:209, 1993.

[25] Mike Ross. Size does matter: continuous size estimating and tracking. *QSM-Quantitative Software Management. Citeseer. Disponível em http://www. qsm. com. Acessado em Novembro*, 2007.

[26] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.

[27] Thomas L Saaty. What is the analytic hierarchy process? In *Mathematical models for decision support*, pages 109–121. Springer, 1988.

[28] Norman F. Schneidewind. Body of knowledge for software quality measurement. *Computer*, 35(2):77–83, 2002.

[29] Richard W. Selby and Adam A. Porter. Software metric classification trees help guide the maintenance of large-scale systems. In *Proceedings. Conference on Software Maintenance*, pages 116–123. IEEE, 1989.

[30] Omar Shatnawi and Hussein bin Abdullah. Discrete time nhpp models for software reliability growth phenomenon. *The International Arab Journal of Information Technology*, 2009.

[31] Per E. Strandberg, Wasif. Afzal, and Daniel Sundmark. Decision making and visualizations based on test results. In *International Symposium on Empirical Software Engineering and Measurement, (ESEM)*, 2018.

[32] Ryouei Takahashi, Yoichi Muraoka, and Yukihiro Nakamura. Building software quality classification trees: Approach, experimentation, evaluation. In *Proceedings The Eighth International Symposium on Software Reliability Engineering (ISSRE)*, pages 222–233, 1997.

[33] Jeff Tian. Integrating time domain and input domain analyses of software reliability using tree-based models. *IEEE Transactions on Software Engineering*, 21(12):945–958, 1995.

Appendices

Appendix A
Popular Science Summary

Mjukvaruföretag har ofta som mål att bara leverera ny programvara till kunder. När det är gjort känner man sig nöjd och sen börjar man på nästa programvara. Men hur bra den föregående faktiskt var vet man inte om inte kunden ger några synpunkter eller respons och även det är lite information. Kan man på något sätt stanna upp lite och ta reda på hur bra en programvara var på ett sätt som ändå gynnar företaget.

Vid släpp av programvara så sammankommer uppdateringar från en uppsjö av avdelningar. Varje avdelning har sina egna tester och diverse data, som ger dem en känsla av hur bra deras egna artiklar är. Men det finns inget smidigt sätt att kolla på data från flera avdelningar för att få en bild av kvaliteten på programvaran. Vidare finns det inga konkreta definitioner på vad en kvalitativ programvara är. Till exempel, finns det kodmått eller mätvärden som måste vara med och väger vissa mer än andra?
Detta examensarbete undersöker hur man genomför en kvalitetsmätning av programvara inom Axis Communications. Syftet är att skapa en modell för att automatiskt estimera programvarukvalitet. Med hjälp av data som finns tillgänglig innan man slappt programvaran så ska modellen ge indikationer på hur den kommer bete sig i fält.
Personal ansvariga för utsläpp av programvara är beroende av data för att göra beslut. Deras beslut påverkar direkt kundens upplevelse och därmed hela företagets levnadsbröd. Med tillförlitlig data så kan man ge kunden en stabilare produkt. Det blir även enklare för företaget/avdelningen att planera arbetet framöver då de ser var de ligger och eventuellt vilka åtgärder som behöver tas. Det finns även en möjlighet att förkorta processen av utsläppet. Med en bra modell så kan man ta snabbare och säkrare beslut.
Med våra resultat så behöver de inte fatta besluten helt på egen magkänsla utan kan falla tillbaka på hård data. Dessutom så kommer modellerna med tiden, allteftersom de får fler datapunkter, tränas bättre och ge mer precisa estimeringar av kvaliteten.
Vår metod bestod i princip av tre stora steg. Först gjordes en undersökning av litteratur om vilka mätvärden som kan vara relevanta att ha med, samt en undersökning inom företaget om vilka mätvärden och data som finns tillgängliga.

Det andra steget var att extrahera datan på ett sätt som gör att vi kan använda den i vår modell. Datan som vi valde att gå vidare med baserade vi främst på diskussioner och intervjuer vi haft med relevant personal på företaget, samt litteratur vi hade hittat.
Till sist så skapade vi flera modeller med olika kombinationer av datan. Till exempel så använde en modell bara ett kodmått, en annan modell använde tre o.s.v. Dessa evaluerade vi genom att träna upp modellerna på alla utom en datapunkt och sedan estimera på den borttagna datapunkten. De tre bästa presenterades för företaget.
För att skapa modellerna så behövdes definitioner fastställas, bl.a. vad kvalitet är. Linjär regression användes för att hitta samband mellan de tillgängliga mätvärdena. Från detta skapades modeller som baserat på indatan svarar på frågan hur bra programvaran är.

Appendix B
Metrics included in the research

Axis product metrics	Code metrics	Testing Metrics:	Process Metrics: amount of docum	General quality metrics	Performance Metrics:
Boottime	Program size	Service request time	# Tickets	Portability	Scaling with problem size
CPU usage	System size	Defect find rate	# Commits	Sustainability	Build time
Flash usage	Binary size	Mean time to repair	# Products	Customer satisfaction	Compile time
Memory available	Lines of code	Test failures	# FW Images	Scalability	Execution time
System ready time	Code turmoil/churn, changing/growing rate	# Bug fixes compared to history	# Features	Reliability	FLOPS (flyttalsoperation per sekund)
Response time	# Classes	# Bugs	Requirements Change Request	Portability	
Package changes	# Modules	Defect density (#defects/size of the software)	# Developers	Formal correctness	
Release blockers	Cyclic dependency	Critical problems prior to release	Story points/features	Interoperability	
Master prio	Cyclomatic complexity	Open defects	Development manhours	Maintainability	
QA test data	Complexity of source code	# Tests	Cycle time (software eng. process)	Accuracy	
	Couplings	# Passing tests	Working hours/story	Code language	
	Clarity, (Information entropy)	Code coverage	Feature delivered	Technical debt	
	Depths (inheritamce, statements, func. ca	# Platforms covered by tests	Functionality	Testability	
	Duplicate code	Usage- profiles	# Documentation files		
	Weighted method count	Reproducibility	# Program launches		
	Comment density/code to comment ratio	# Assertions	Documentation		
	Total statement count (executable code)	# Bugs found / # Estimated	GitHub		
	# Comments	Collateral damage (bugs by fixing)	JIRA to track development		
	# Non-commented LOC		Productivity		
			Recursive validation		
			Redmine project management		
			# Authors/committers		
			Request count		

Figure B.1: Matrix of all metrics we found

Axis product metrics	Code metrics	Testing Metrics:	Process Metrics: amount of document	General quality metrics	Performance Metrics:
Boottime	Program size	# Bugs	# Tickets	Portability	Scaling with problem size
CPU usage	System size	Defect find rate	# Commits	Sustainability	Build time
Memory available	Binary size	Mean time to repair	# Products	Customer satisfaction	Compile time
System ready time	Lines of code	Test failures	# FW Images	Scalability	Execution time
Package changes	Code turmoil/churn, changing/growing rate	# Bug fixes compared to history	# Features	Reliability	FLOPS (flyttalsoperation per sekund)
Release blockers	# Classes	Service request time	Requirements Change Request	Portability	
Flash usage	# Modules	Defect density (#defects/size of the	# Developers	Formal correctness	
Response time	Cyclic dependency	Critical problems prior to release	Story points/features	Interoperability	
Master prio	Cyclomatic complexity	Open defects	Development manhours	Maintainability	
QA test data	Complexity of source code	# Tests	Cycle time (software eng. process)	Accuracy	
	Couplings	# Passing tests	Working hours/story	Code language	
	Clarity, (Information entropy)	Code coverage	Feature delivered	Technical debt	
	Depths (inheritamce, statements, func. call	# Platforms covered by tests	Functionality	Testability	
	Duplicate code	Usage- profiles	# Documentation files		
	Weighted method count	Reproducibility	# Program launches		
	Comment density/code to comment ratio	# Assertions	Documentation		
	Total statement count (executable code)	# Bugs found / # Estimated	GitHub		
	# Comments	Collateral damage (bugs by fixing)	JIRA to track development		
	# Non-commented LOC		Productivity		
			Recursive validation		
			Redmine project management		
			# Authors/committers		
			Request count		

Figure B.2: Matrix of the final metrics selection

Appendix C
APH poll

Which is more important?

Metric A	<<<<	<<<	<<	<	o	>	>>	>>>	>>>>	Metric B
number of products										pulled products
number of products										delay
number of products										total tickets
number of products										bug tickets
number of products										priority
number of products										severity
pulled products										delay
pulled products										total tickets
pulled products										bug tickets
pulled products										priority
pulled products										severity
delay										total tickets
delay										bug tickets
delay										priority
delay										severity
total tickets										bug tickets
total tickets										priority
total tickets										severity
bug tickets										priority
bug tickets										severity
priority										severity
Severity	<<<<	<<<	<<	<	o	>	>>	>>>	>>>>	
showstopper										severe
showstopper										medium
showstopper										small
severe										medium
severe										small
medium										small
Priority	<<<<	<<<	<<	<	o	>	>>	>>>	>>>>	
high										medium
high										low
high										incoming
medium										low
medium										incoming
low										incoming