Empirical Analysis of Hidden Technical Debt Patterns in Machine Learning Software

Master's thesis in Software Engineering

MOHANNAD ALAHDAB

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Empirical Analysis of Hidden Technical Debt Patterns in Machine Learning Software

MOHANNAD ALAHDAB

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Empirical Analysis of Hidden Technical Debt Patterns in Machine Learning Software
MOHANNAD ALAHDAB

Supervisor: Gul Calikli, Department of Computer Science and Engineering
Examiner: Robert Feldt, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Division of Software Engineering
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Abstract

Development, deployment, and maintenance of Machine Learning (ML) based software products are costly. However, these costs are usually neglected. Challenges regarding maintainability of ML software were explained under the framework of "Hidden Technical Debt" (HTD) by Sculley et al. [10] by making an analogy to technical debt in traditional software. HTD patterns are due to a group of ML software practices and activities leading to the future difficulty in ML system improvements, many unhandled errors in the long term and hence considered as main causes of the increase in maintainability cost. Moreover, some of those patterns keep expanding unnoticed; for that reason, they are called *hidden* patterns. ML systems have a special ability for increasing technical debt due to ML specific issues at the system level in addition to having all the problems of regular code. The aim of this thesis is to empirically analyze which and how HTD patterns emerge during the early development phase of ML software, namely the prototyping phase. For this purpose, we conducted a case study to analyze ML models. These models will go into production and then integrated to the software system owned by Västtrafik, that is the public transportation agency in the west area of Sweden. In order to investigate the generalizability of our case study findings, we conducted a workshop with practitioners consisting of data scientists and software engineers. During our case study, out of 25 HTD patterns, we were able to detect 12. One of the 12 patterns detected was only observed to a limited extent. Observed patterns during prototyping are mainly underutilized data dependencies (e.g., correlated, bundled, and ϵ features) and ML code smells (e.g., glue code, pipeline jungles, dead experimental code paths). We also observed entanglement, configuration debt, abstraction debt, prediction bias, multiple language smell, data testing debt, and cultural debt up to some extent. All of the 12 HTD patterns that were undetected, could only be detected after deployment of ML software. The only undetected HTD pattern is "Plain Old Data Type Smell", since we did not implement the ML algorithms from scratch, but instead used existing ML libraries owned by an online cloud solution. Our workshop results indicate that, majority of our findings are applicable to other ML application domains. Practitioners also agreed that prototypes built by data scientists are not ideal in terms of software engineering (SE) practices. Hence, developers need to refactor the prototypes in order to prepare for the production stage.

Keywords: Technical Debt, Hidden Technical Debt Patterns (HTD) Patterns, Machine Learning (ML), Software Engineering (SE), Maintainability, Feature Engineering.

Acknowledgements

First of all, I would like to thank my thesis supervisor Dr. Gül Calikli of the Software Engineering Divison at Chalmers | University of Gothenburg. The door of Dr. Calikli office was always open. She has been supporting me from the first moment and she was always ready to answer all my questions at any time. She guided me in the right direction and she never ever held me back for any information. I would also like to thank Prof. Robert Feldt of the Software Engineering Divison at Chalmers University of Technology as the examiner of this thesis, and I am gratefully indebted to him for his very valuable advice to produce a very good thesis with high quality.

Of course, I would like to thank my family mom, dad, my three sisters and my friends who supported me and encouraged me all the time.

Mohannad Alahdab, Gothenburg, May 2019

Contents

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**

1.1 Background 1
1.2 Purpose of the study 3
1.3 Research Questions 4
1.4 Limitations and Delimitations 4
1.5 Report Structure 5

2 Related Work and Background **6**

2.1 State of the Art: From ML prototypes to large scale ML intensive software products 6
2.2 Background Information 9
2.2.1 Technical Debt In Traditional Software 9
2.2.2 Hidden Technical Debt in ML Intensive Software 10

3 Case Study Methodology **17**

3.1 Objective 17
3.2 The case 17
3.2.1 Data Collection 18
3.2.2 Technology and Implementation 18
3.3 Research questions 18
3.4 Methods 19
3.4.1 Data analysis and preprocessing 19
3.5 Building ML Models 20
3.5.0.1 Implementation Details 21

4 Case Study Results **26**

4.1 Results for Developed ML Models 27
4.1.1 Observed HTD Patterns 30
4.2 Projections for the Final Software System 43
4.2.1 Impact of Observed HTD Patterns on the Software System . . 44
4.2.2 HTD Patterns to be Observed on the Deployed Software System 45
4.3 Unobserved HTD Patterns 48

5 Interactive Workshop Methodology and Results 49
5.1 Workshop Aims . . . 49
5.2 Workshop Design . . . 49
5.3 Workshop Result . . . 52
5.3.1 Practitioners' Prioritization of HTD Patterns and the Rationale Behind . . . 53
5.3.2 Practitioners' Recommendations to Manage HTD Patterns . . 56

6 Threats to Validity 59
6.1 Construct Validity . . . 59
6.1.1 Case Study . . . 59
6.1.2 Workshop . . . 59
6.2 Internal Validity . . . 59
6.2.1 Case Study . . . 59
6.2.2 Workshop . . . 60
6.3 External Validity . . . 60
6.3.1 Case Study . . . 60
6.3.2 Workshop . . . 61
6.4 Dependability . . . 61

7 Conclusion and Future Plan 62
7.1 Conclusion . . . 62
7.2 Future Plan . . . 65

Bibliography 67

A Data Collection Of The Selected Features I
A.1 The numerical weather summary data description . . . II
A.2 The numerical values for the day after and before the holiday . . . III
A.3 The numerical values for the day type . . . III
A.4 The technical pipeline and components of ML model development . . III
A.5 Execute Python Script component used for preparing the input features. IV

List of Figures

2.1 Resources that are frequently needed by AI systems [17]. 7
2.2 Complete loop for effective contextual learning, showing the four system abstractions we define [18]. 8
2.3 The technical debt landscape [11] 9
2.4 Four colors in a backlog.The element areas reconcile four types of possible improvements [11] 10
2.5 ML model configuration options 12
2.6 Correction cascades layers and dependencies [22]. 13
2.7 Hidden feedback loops online bookstore. 15
2.8 Online bookstore with fixed thresholds in dynamic systems. 15

3.1 ML model development workflow design. 21
3.2 Azure ML pipeline implementation for ML models. 22
3.3 "Apply SQL Transformation" component for preparation dataset. . . 23
3.4 Example of "Apply SQL Transformation" component to split the data as training dataset. 23
3.5 Example of "Apply SQL Transformation" component to split the data as testing dataset. 23
3.6 "Edit Metadata" component configuration for preparation dataset. . . 24
3.7 The default configurations of the boosted decision tree regression ML algorithm component. 24
3.8 The default configurations of the decision forest regression ML algorithm component. 25

4.1 The correlation between the features, mean absolute error and coefficient of determination with boosted decision tree regression algorithm. 28
4.2 The correlation between the features, mean absolute error and coefficient of determination with decision forest regression algorithm. . . . 29
4.3 Features correlation map for Delsjön parking the four quarters of the year 2014. 32
4.4 Features correlation map for Delsjön parking the four quarters of the year 2015. 33
4.5 Glue code and pipeline jungles in the experiments. 35
4.6 The distribution of the actual and the predicted dataset for one year. 37
4.7 The distribution of the actual and the predicted dataset for one month. 38
4.8 The distribution of the actual and the predicted dataset for one year. 39
4.9 The distribution of the actual and the predicted dataset for one month. 40

4.10 The distribution of the actual and the predicted dataset for one year. 41
4.11 The distribution of the actual and the predicted dataset for one month. 42
4.12 Expected ML prototype on the final software system with the potential features. 45

5.1 HTD pattern $100 questionnaire (round 1). 51
5.2 HTD pattern $100 questionnaire (round 2). 52

A.1 Azure technical pipeline and components for ML model pipeline . . . III
A.2 Execute Python Script component used for preparing the input features. IV

List of Tables

3.1 Parking lots data set 18
3.2 List of the potential features 20
3.3 Azure components of pipeline implementation for ML components. . . 22

4.1 Feasible triples for a highly variable Grid 27
4.2 The result of adding features iteratively by using boosted decision tree regression algorithm. 28
4.3 The result of adding features iteratively by using decision forest regression algorithm. 29
4.4 The result of adding B, C and D features by using decision forest regression algorithm. 30
4.5 The result of adding B, C and D features by using boosted decision tree regression algorithm. 30
4.6 Results of ML algorithms which trained with all features for the whole data set of 2014 and tested by the year 2015 of historical parking events. 36
4.7 Results of ML algorithms which trained with all features for the whole data set of 2014 2015 and tested by the year 2016 of historical parking events. 39
4.8 Results of ML algorithms which trained with all features for the whole data set of 2014, 2015, 2016 and tested by the year 2017 of historical parking events. 41

5.1 Workshop practitioners details. 52
5.2 Workshop practitioners results for $100 questionnaire (First Round). 53
5.3 Workshop practitioners results for $100 questionnaire (Second Round). 53

1 Introduction

1.1 Background

Nowadays, Machine Learning (ML) algorithms are not only being implemented in research labs, but they have started to become an integral part of software products, resulting in Machine Learning (ML) software. Many ML applications are common nowadays in our lives in the form of software products including recommender systems (e.g., Netflix [1], LinkedIn [2]) and speech recognition systems (e.g., Apple Siri). Social media platforms such as Facebook develop ML applications for ranking posts in the news feed, speech recognition, text translation as well as real-time photo and video classification [16]. Moreover, companies such as Volvo, Tesla, and Google are running tests on self-driving cars, and experts expect that people will start to use self-driving cars soon. As a result of these advances, as also indicated by Martinez-Plumed et al. [17], it has become obvious that the prediction performance of ML algorithms is not the only metric we should focus on.

Since ML intensive software also has a life cycle as the traditional software does, the emergence of ML intensive software, also brings the challenges regarding the maintainability. Maintainability of traditional software products is still a challenge, and technical debt is an obstacle in the way of software maintainability. The "technical debt" expression was first coined by Ward Cunningham twenty years ago and it is a software concept that tries to clarify the implicit cost of more modifications caused by selecting a simple solution now instead of doing a better concept that will take longer [11].

However, existing practices, tools, and techniques to tackle technical debt are not adequate to overcome the challenges related to the maintenance of ML intensive software. This is because the implementation of ML algorithms is quite different compared to how traditional software is implemented. Some differences between them are: traditional software mostly consists of a set of commands that are implemented by the developer so that the computer can follow and execute these instructions. On the other hand, during the implementation of ML algorithms, instructions are not explicitly implemented by the developer. Instead, some constraints on the behavior of the program are provided in the form of a dataset of input-output pairs where the program learns what to do based on that data.

Machine learning allows the developer to work fast and the results can be delivered quickly, but in the long run, it becomes a challenge to maintain ML intensive software [10]. As a result of the experience gained through the implementation of ML algorithms integrated into software products [3, 4], the challenges of software engineering ML intensive systems have become obvious. Sculley et al. [10] coined the term "hidden technical debt" to address challenges in the maintainability of ML intensive software by making an analogy to the concept of technical debt in traditional software [10]. "Hidden" aspect of technical debt in ML systems is due to the fact that such technical debt causes gradual changes in the system that are not immediately visible, and hence very hard to detect or debug. Sculley et al. [10] state that the existing tools, practices, and methods used to tackle technical debt in traditional software are not adequate to tackle hidden technical debt in ML intensive software and these are some of the reasons:

1. Complex Models Erode Boundaries
In traditional software, it is common to detect and isolate a bug. However, in ML software, changing one thing changes everything and it is hard to make local changes. Examples for this category are: entanglement, correction cascades and undeclared consumers.

2. Data Dependencies Cost More than Code Dependencies
There are many static analysis tools to detect code dependencies. In contrast, there is a lack of appropriate tools to identify data dependencies. Moreover, data behavior is unstable over time and that is related to feature engineering, where the importance of these features could be changed in the long term.

3. Feedback Loops
Many ML models depend on their outputs to adjust the training data model in the long term. However, this feedback loop makes the model isolated. In addition, large-scale ML systems contain many ML models that are working together and depend on each other. This type of feedback loop is not easy to detect and it could cause many hidden problems.

4. ML-System Anti-Patterns or System-level Spaghetti
It may seem that developing an ML pipeline model does not have many traditional programming components but the fact is that there are many stages in ML development such as training data preparation that use many different traditional programming languages. Examples for this category are glue code, pipeline jungles, dead experimental code paths, and common smells.

5. Configuration Debt
The large-scale system has a wide range of configurable options, such as selected features which are used, nature of the data set, algorithm-specific learning settings, verification methods, etc.

6. Dealing with Changes in the External World
After you build the model, the external world changes. Hence, you have to tweak the weight of the features to the most recent data in order to retrain a new model.

7. Other Areas of ML-related Debt
The managerial part also plays a major part of ML product development. This category has also many debts for example, data testing debt which is related to data preparation, reproducibility debt which is related to how many experiments should be run, process management debt which is related to analysis of the debts of running many ML models at the same time as large and mature ML system and finally, the cultural debt which is related to the skills and specialists differences between the team members who will develop ML system.

Therefore, it is possible of using HTD patterns which have been mentioned by Sculley et al. [10] as a framework to detect then eliminate those patterns during developing and deploying ML intensive software. Consequently, the maintainability cost will be considered from the beginning and early stage of ML software development.

1.2 Purpose of the study

The purpose of this study is to empirically analyze HTD patterns in the early phase of ML intensive software development, namely "prototyping phase". Early detection of HTD patterns in ML software systems is crucial, since they cause gradual changes in the systems that might not be immediately visible, and hence very hard to detect or debug. If those HTD patterns are detected in later stages of ML software development, it might be quite costly and infeasible to remove them in order to ensure maintainability.

Empirical analysis of HTD patterns in ML prototypes provides information to develop methods to detect and remove them. Moreover, taking into account the system where ML applications will be integrated is crucial to figure out the following: (1) the extent up to which the overall system will be affected due to the existing HTD patterns, so that one can prioritize which HTD patterns should be removed before deployment; (2) HTD patterns which do not exist in the current prototype, but that will certainly show up after deployment of the ML system. Therefore, we conducted a case study, where we first built ML prototypes and then investigated existing HTD patterns in those prototypes. Such effort also resulted in reporting how we detected them as well as proposing methods to remove them whenever possible. In our case study, we built ML models for empty parking lots prediction to be integrated into the software system owned by Västtrafik. Since our case study was conducted for this specific case, in order to investigate generalizability our results and complement our findings wherever possible, we conducted a workshop with practitioners consisting of data scientists and software engineers.

1.3 Research Questions

In this research, we will try to answer the following research questions by employing two research methodologies, which consist of a case study followed by a workshop. The research questions are:
RQ1 : Which technical debt patterns are observed during early phases of ML software development lifecycle (i.e., prototyping phase)?
RQ2: What is the perspective of practitioners to HTD patterns?

- **RQ2.1**: How practitioners prioritize HTD patterns?
- **RQ2.2**: What is the rationale behind this prioritization?

RQ3: What are the methods/techniques, which practitioners recommend or use to manage HTD?

We use the results we obtained from the case study to answer RQ1, while we aim to answer RQ2 and RQ3 by analyzing the qualitative and quantitative data we collected during the workshop that we conducted with practitioners.

RQ2 aims to investigate the relevance and importance of the HTD patterns we could and could not detect during our case study, to practitioners based on their experience. Answering RQ2 will help us assess the generalizability of our results while answering RQ3 will help us find out whether methods/techniques we employed to detect/remove HTD patterns can be complemented by those of practitioners.

1.4 Limitations and Delimitations

This thesis context is in a specific domain, which is an empty parking lots prediction that aims to predict a total count of empty parking lots for a given date and time slot. These are the expected limitations:

- The dataset provided by Västtrafik is for the period 2014-2017. In our case, the prediction system will be a standalone tool which does not consume any data continuously generated by other systems. Thus, regarding HTD due to data dependencies, this study will focus on underutilized data dependencies pattern such as bundled features and ϵ-features.
- The subject and the observer are the same person, who is a software engineer with six years of experience in the industry, is developing the prototypes of prediction models.
- The used tool is only one tool which is Microsoft Azure Machine Learning Studio. However, we focus on the concept of technical debt that can be observed in any ML based software regardless of the technology used for deployment.
- In our case, not all HTD patterns, which have been mentioned in Sculley et al. [10], are observed. This depends on many aspects like the case situation, the context, the data behavior, and the development environment. Another important reason that the current stage covers the development of ML prototypes but HTD patterns, which are not observed currently, may appear in the later stages of ML software development lifecycle.

In the near future, few ML intensive systems will implement ML algorithms from scratch. Instead, for such systems training will take place in the cloud using the main API and libraries. Most companies do not have enough human resources having the skills required to design and implement ML algorithms. Therefore, there is a need to provide developers APIs that allow them to embed ML functionalities into software applications. For that reason, companies have decided to simplify the procedure of building ML models by offering libraries (e.g., TensorFlow, etc.), framework and also on-line cloud solution such as Google, Amazon, IBM, and Microsoft Azure studio. They have offered many facilities like a friendly user interface, drag and drop options, importing different types of datasets and other services. Moreover, ML in the cloud systems is cheap to operate in terms of hardware and software. Hence, using a service such as Microsoft Azure Machine Learning rather than implementing the ML algorithms from scratch is not a bug, but instead a feature in the design and implementation of this case study.

Further limitations will be discussed in "Threats to Validity" section 6.

1.5 Report Structure

The rest of this thesis is structured as follows: Chapter 2 describes the related work and gives background information. Chapter 3 explains the research methodology we followed to conduct the case study. Chapter 4 presents and interprets results of the case study. Chapter 5 explains how we designed, prepared and conducted the workshop with the software practitioners as well as summarizing the results obtained. Chapter 6 addresses threats to validity. Finally, Chapter 7 concludes the thesis and mentions future work.

2

Related Work and Background

This section starts with state of the art, explaining the very recently emerging research to tackle software. It also explains the differences between traditional and ML software. Moreover, the literature review explains in detail HTD patterns which were identified by Sculley et al. [10] that will help the reader to a better understanding of the technical debt and its impact in ML software. These HTD patterns will guide the software engineers to make a good indicator from two sides, which are the expected maintainability cost and the readiness of ML software as a real product.

2.1 State of the Art: From ML prototypes to large scale ML intensive software products

As a result of the increase in computing power and the amount of data collected through the internet [7], ML researchers and industry have started collaborating to produce commercial ML software products for the regular user. They are trying to steer ML, which is a branch of Artificial Intelligence (AI), from the pure research labs to the commercial market. There are many examples of AI applications which are used now in daily life such as Virtual Personal Assistants (e.g., Microsoft Cortana, Apple Siri, Amazon Alexa and Google Home) [15]. In [15], the authors discuss the new model of VPAs which will be utilized to develop the communication and the interactions between the people and the machines by employing many integral technologies, for instance, gesture recognition, image/video recognition, speech recognition, the vast dialogue and conversational knowledge base, and the general knowledge base. Also, more recent attention has focused on ML usage in social media which has become an integral part of our lives. A particular study by Hazelwood et al. [16] have presented applied ML at Facebook [16]. At Facebook, ML gives users the ability to interact with other users through covering many services like ranking posts for news feed, speech recognition and text translations, and photo and real-time video classification. Besides the Facebook Ads for different age group and according to user behavior, Facebook also offers the search launches which give the user the ability to search through different materials, e.g., videos, photos, people, events, etc. Moreover, the Facer is Facebook's face detection and recognition framework [19].

Researchers so far have focused on the prediction performance of ML algorithms. However, as ML algorithms have become an integral part of large-scale software products, it has become obvious that prediction performance is not the only metric one should focus on. In recent years, researchers have started to address the importance of development, testing, deployment, and maintenance of ML intensive software [17]. Martinez-Plumed et al. [17] have tried to explain the resources of the costs of developing and deploying AI software systems such as nature of data, knowledge base, human faults, computing frequencies, software, hardware, and network facilities and resources, development time like training and testing practices, etc. In general, the measurement of ML performance such as the accuracy or the error percentage is the target for ML specialists. Martinez-Plumed et al. [17] indicate that the measurement will be more accurate with more generalization and comprehensive measures of resources. Among the resources the authors mention are those related to software (e.g., ML algorithms and libraries) as well as the content of the data provided as input to ML algorithms and the physical time needed for the collection of that data (i.e., waiting/input times, iteration cycles, etc.).

Part of the developing costs is paid just once in the initial development stage, whereas other costs would be acquired for any application updates or upgrades. This study has also analyzed the required internal and external resources of the AI system life cycle. Martinez-Plumed et al. [17] have discussed one example, which is speech recognition, and the main artifacts of this AI system have been shown in Figure 2.1. Figure 2.1 is also an example of the importance of development and deployment, showing the fact that experimentation and prototype implementation is just the beginning stages of the ML intensive software development.

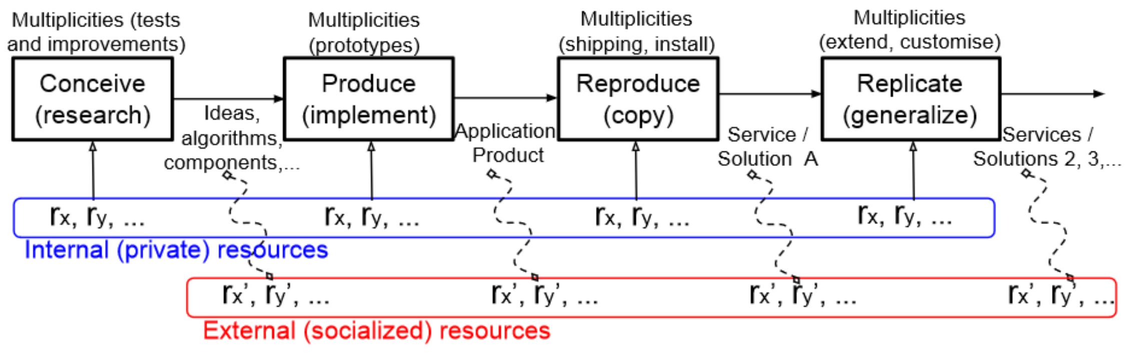

Multiplicities (tests and improvements)
Multiplicities (prototypes)
Multiplicities (shipping, install)
Multiplicities (extend, customise)
Conceive (research)
Produce (implement)
Reproduce (copy)
Replicate (generalize)
Ideas, algorithms, components,...
Application Product
Service / Solution A
Services / Solutions 2, 3,...
rx, ry, ...
Internal (private) resources
rx', ry', ...
External (socialized) resources

Figure 2.1: Resources that are frequently needed by AI systems [17].

During the development and deployment of ML intensive software such as online advertising systems, experts at Google gained experience in how to engineer such systems [3]. All these experience gained led D. Sculley and his colleagues at Google to come up with a framework, which they termed as "HTD [10]". In their work, Sculley et al. [10] make an analogy to the "technical debt" in traditional software and explain how these HTD patterns might emerge in an ML intensive software and how they might negatively affect maintainability of ML intensive software. The same group of Google engineers/researchers later came up with some guidelines for the testing of ML intensive software [12]. Most of the tests they propose in this study are also useful in tackling with HTD.

Referring to HTD patterns identified by Sculley et al. [10], Agarwal et al. [18], proposed a solution to tackle "direct feedback loops", which is HTD pattern that often occurs when the resulting ML system might bias the user's feedback to the system, which in turn, directly affects the selection of users' data for future training of that ML system [18]. HTD pattern "Direct feedback loops" is often observed in ML intensive systems such as online news websites, online advertisements, and technical support assistants. In order to tackle this kind of HTD, Agarwal et al. [18] propose a complete loop for effective contextual learning consisting of the phases of deployment, exploring, logging and learning as shown in Figure 2.2.

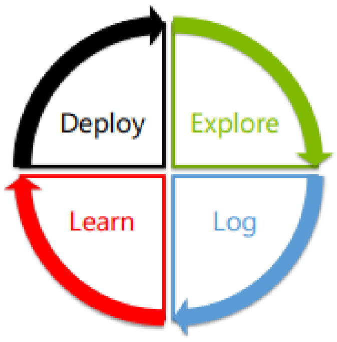


Figure 2.2: Complete loop for effective contextual learning, showing the four system abstractions we define [18].

In their paper, Agarwal et al. [18] have dealt with two live production deployments, which are a content recommendation and machine failure handling as tech support. Moreover, the main technical debts which will be explored in this case study are feedback loops, bias, distributed data collection, changes in the environment, and weak monitoring and debugging. Agarwal et al. [18] have mentioned HTD framework, which was proposed by Sculley et al. [10]. They have addressed by design the challenges that cause difficult-to-debug performance degradation while developing the decision service and later in the deployment, consequently lead to the technical debts. The main four sources for the failures are (F1) partial feedback and bias, (F2) incorrect data collection, (F3) changes in the environment and (F4) weak monitoring and debugging.

Regarding research and engineering efforts for software engineering ML intensive systems, there are also studies in the literature for testing ML software. Besides the study by Sculley and colleagues at Google [12], there are some studies such as automated white box testing of Deep Learning (DL) systems [20], automated testing of DL software running on autonomous cars [21] and SADL testing of DL systems by using test surprise scenarios adequacy criterion which is called Surprise Adequacy for Deep Learning Systems [24]. Another study has discussed the main challenges for developing DL systems [25].

Collectively, these studies outline a list of proposed hidden costs and obstacles for developing and deploying ML products where Sculley et al. [10] have summarized those patterns throughout "The High-Interest Credit Card of Technical Debt" [10]. Sculley et al. have analyzed the development and deployment of ML products from another angle. In their opinion, it is risky to think of producing quick and powerful ML solution is for free and without any side effects. They identified ML technical debt patterns which we will explain and investigate in section 2.2.2.

2.2 Background Information

2.2.1 Technical Debt In Traditional Software

In general, companies try to balance between the quality and the required resources like cost and time during the software development process. However, some of these software companies deliver sub-optimal solutions in order to shorten the time-to-market, or they have a lack of resources by implementing "quick fixes or cutting corners" during software production [13]. Consequently, these sub-optimal solutions in the software steadily increase; by the time the fixed code will deeply be part of the total solution. Now when a new feature is required to be added, the system will extend, and consequently these sub-optimal solutions will delay or disrupt the software development process. The previous result of disrupting software development is called Technical Debt (TD) [13].

TD Categorization Landscape

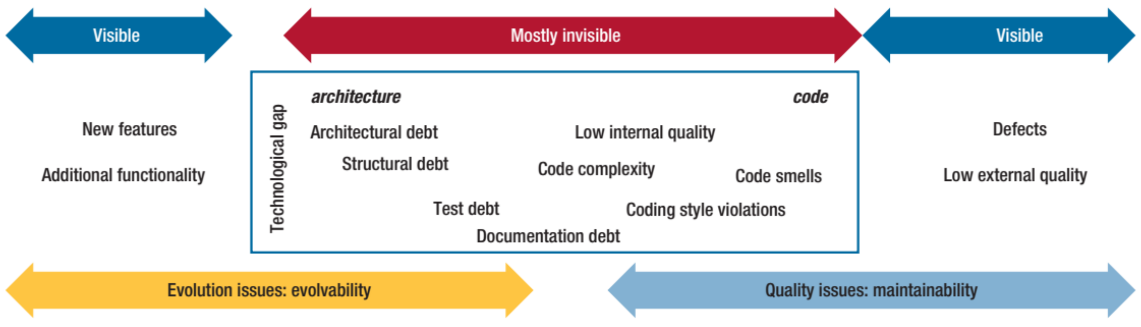


Figure 2.3: The technical debt landscape [11]

Figure 2.3 illustrates a possible organization of technical debt, but that will depend on the given case. As has been shown, this figure is divided into three main areas. The part on the left side represents the project evolution challenges like adding a new feature. The part on the right side displays the quality section on both sides the internal and external. The box in the middle illustrates the potential technology gap which is invisible technical debt with respect to the quality and evolution [11].

Tackling Technical Debt
"Schedule Pressure" is one of the most important reasons for the technical debt according to many of interested authors in this field. There are many software development managerial styles such as agile and waterfall. The agile developers suppose that they are using an iterative development process. They believe that they are quite unaffected by technical debt [11]. Although agile methodology supports the rapid implementation solution, at the same time it sometimes doesn't have clear

systematic testing. Besides, it has no time for proper design, and that will cause a large amount of debt rapidly. Now, the main question is how to tackle technical debt or try to decrease counting too much of it.

The first step "awareness" is listing and defining the potential debt and its causes. The second step "things to do" is to analyze this debt, which consists of finding the correlations between the debt and the related tasks during the deployment process and planning. Figure 2.4 displays the distribution of the suggested potential improvements. "The tasks to attend to in the future to increase value, such as adding new features (green) or investing in the architecture (yellow), and to reduce the negative effects on the value of defects (red) or technical debt (black)" [11].

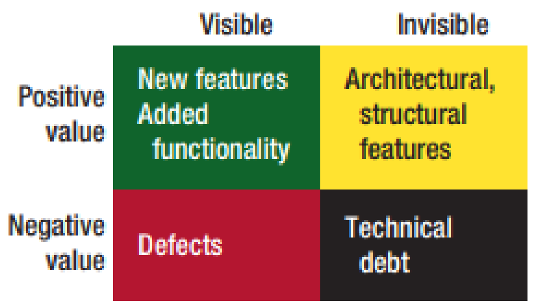

Visible
Invisible
Positive value
Negative value
New features Added functionality
Architectural, structural features
Defects
Technical debt

Figure 2.4: Four colors in a backlog.The element areas reconcile four types of possible improvements [11]

2.2.2 Hidden Technical Debt in ML Intensive Software

This section will explain HTD patterns in ML intensive software systems [10]. During developing the ML software prototypes during our case study, some of HTD patterns were observed, but others were not. We will explain each group with a brief explanation for each pattern.

Observed HTD Patterns

- **Entanglement**
  ML algorithms combine different signals, entangling them in a way that makes the development process not easy. Analyzing the system behavior with this complex combination of signals is not a trivial task. The primary measurement in this pattern is the accuracy rate and the error percentage, where many dimensions could play a significant role in this measurement. Examples of these aspects are: adding new features, removing some features, using hyperparameters, adjusting the learning settings, sampling methods, and determining fuzzy or crisp thresholds. Many strategies are used to avoid this destructive pattern. The first one is model isolation, which makes the tracking of the prediction measurement with respect to changes in the previously mentioned aspects. The second one is to use helping tools like visualization to detect the changes quickly in prediction behavior. The third solution is to use ensemble models to increase the robustness of ML systems against entanglement.

- **Bundled Features and ϵ-Features**
"Bundled features" are a group of features that have been added to the model without analyzing their impact on the model accuracy performance, and probably they do not have that impact, and they may cause many problems in the accuracy at the same time. Each of the features, which altogether make up the bundled features, are ϵ-features. Sometimes and especially in the initial stage, we prepare a list of potential features which depends on the context and expectations about features' effectiveness. The relationship between the features plays a significant impact on the performance as we will see in the Correlated Features section. Besides, bundled features and ϵ-features patterns are considered as underutilized data dependencies.

- **Correlated Features**
Regarding the correlated features, we can discuss three cases. The first one is the strongly correlated features with significant impacts on the model performance where this case is the optimal situation for the training ML model. The second one is the strongly correlated features which have no significant effect on the model performance. Finally, the third one is the critical case where there is a strong correlation, and one of them has a substantial impact, and another one has a slight effect or does not have a significant impact on the prediction performance of the ML model. The third one is not easy to manage until discovering the useful features within the consideration of the features' impact over time. The suggested strategy to avoid this pattern is to apply feature selection techniques to find out which features have a significant effect on the outcome to be predicted and analyze the correlation between input features as explained in a later chapter. In addition, this pattern belongs to underutilized data dependencies HTD patterns category.

- **Glue Code**
ML pipeline contains multiple ML and software components, and it uses glue code to join those components. For instance, data preparation uses software components to prepare friendly-data input for training ML algorithm component. In this case, we can use the traditional strategy to manage this pattern like code refactoring and software optimization. Also, it is a good solution to use a web-based solution like consuming data from API instead of using the static code to collect them.

- **Pipeline Jungles**
A pipeline jungle debt is a particular case of glue code. This concept may appear in the data preparation stage. Pipeline jungles usually used to prepare ML friendly format for ML algorithm components. As it has been mentioned for glue code, before to control this kind of debt, it is required to use the traditional technical debt optimization techniques.

- **Dead Experimental Code paths**
  This pattern also belongs to the same debt category, which is system level spaghetti, and this type of debt is a result of the previous patterns debt, which are glue code and pipeline jungles. ML project requires many experiments with alternative methods. Therefore, there are many glue code and pipeline jungles with accumulated code paths as a result over time.

- **Abstraction Debt**
  Determining the abstraction boundaries in ML project is not easy since the ML model has many components and different resources, all of which should work together. Also, this problem will be more complicated, and the boundaries will be almost lost when many ML models are embedded inside one integrated system.

- **Multiple-Language Smell**
  Using different programming languages to implement ML will cause different types of technical debts during the development, testing, and deployment stages. For instance, refactoring is the most popular programming practice in traditional and ML software, and we can imagine the cost which will be paid when we will do refactoring and testing for multiple languages at the same time.

- **Configuration Debt**
  The configuration of ML systems may also cause debt and play the main role in the quality of the output of the model as well. For instance, determining the data, the selected features, the suitable parameters, and so on. Figure 2.5 illustrates the configuration options which could be tuned during developing ML software. Changing any configuration option will affect the final result and may influence the other options in an untraceable manner.

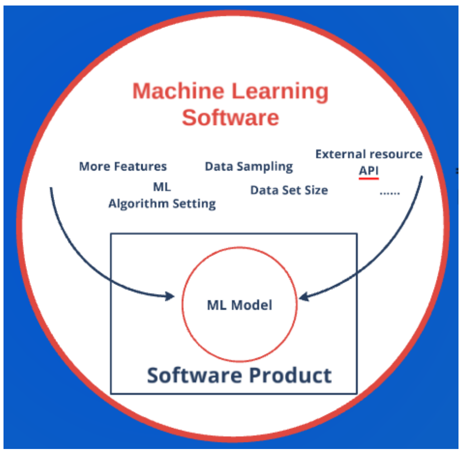


Figure 2.5: ML model configuration options

- **Prediction Bias**
  For a real-time ML system, it is required that the actual, and the predicted

values are almost identical. Prediction bias monitoring is a useful method to eliminate the monitoring and testing debt, so instead of making end-to-end testing for the ML system as a whole, the prediction bias testing is used to monitor the model accuracy step by step against the external factors.

- **Data Testing Debt**
  To build a robust ML project that requires to test the quality of the training and testing input data. Testing the input data quality in ML systems is analogous to testing the code in the traditional one. Therefore data testing is required for any changes in the experiment surroundings situation; consequently, that causes the maintainability cost for any future modification.

HTD Patterns Only Possible to Detect after Deployment

- **Correction Cascades**
  For any ML model, there is usually a required target which we want for the model to learn. Sometimes and for some reasons, the ML outputs do not satisfy the expectations. Therefore, a special filter as a layer could be used to adapt the ML results consequently that will cause a new type of dependency. As we can see in Figure 2.6 [22] there is a layer to prepare a good model and get the required learning expectations. However, all these models become dependent on the original ML system to which correction cascade(s) (e.g., filters, calibrations heuristics, thresholds) have been applied. Any change in the original system will affect the models that are in the form of layers on the top of this original model.

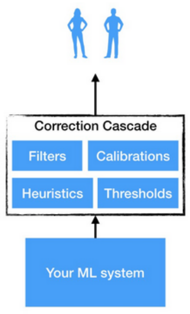


Figure 2.6: Correction cascades layers and dependencies [22].

- **Undeclared Consumers**
  The output of some ML models such as log files or output parameters, will probably be used as input by another system. This type of debt causes a hidden dependency between the model and the other parts of the system. Therefore, if there is any modification in model X and this model generates logs or any type of output files, then model Y which depends on model X and consumes that output will use this output silently. Consequently, the coupling debt between the systems will be increased. The suggested solution to eliminate this type of debt is to define the system's boundaries and identify each

system parameters in a transparent way.

- **Unstable Data Dependencies**
  Today most of the large scale industries use multiple ML models simultaneously for accurate results and better user experience. However, sometimes outputs of ML models are inputs for other models in the same system. Also, in some cases, ML models depend on its output as a feedback loop for the training dataset. The expected result probably will be excellent and satisfy the expectations as long as the behavior of this data is stable. In contrast, the problem will start when this data changes its behavior over time.

- **Legacy Features**
  Legacy features are the features which have been added in the initial stage of developing ML software. These features probably lose their weights because of the redundancy of the features after adding new features during developing ML model. The suggested strategy to avoid this hobo feature is to reevaluate the model over time since the new features may decrease the effectiveness of the initial features. However, the main problem with this strategy is the cost. For example, if the number of features is more than 100 hundreds of features and we want to retrain the model with those features separately to evaluate the model performance sudden increase in running process and in time to train the models. In addition, this pattern is considered an underutilized data dependencies.

- **Direct Feedback Loops**
  Probably, ML model precisely impacts the selection of its upcoming training data. As an example, most of the ML recommendation systems depend completely on the history of the user profile as a training dataset to train the model, and that is called direct feedback loops. In the beginning, displaying the categories and items which the user wanted depending on the user ' behavior history makes the user satisfied. However, over time users' preferences might change, but users get stuck with a selection pool consisting of items provided by the ML system forcing/biasing the users to select items (s)he is not interested in.

- **Hidden Feedback Loops**
  We will use the online bookstore as an example to explain this type of technical debt (see Figure 2.7.). Smart online bookstore uses at least two independently developed ML models, where the first one is to retrieve the suggested books according to the history of the user profile and the second one is to retrieve the books regarding other users' reviews and ratings. Then the final results will be a combination of the outputs from both previous models. Reviews provided by the second ML system might bias the user, as a result of which user might select books (s)he might not be interested in or might not choose books (s)he might be interested in. Since these choices the user makes will be fed into the ML system that makes book recommendations for re-training, the ML-based

book recommendation system will get biased.

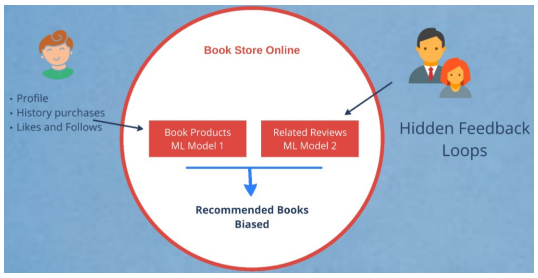


Figure 2.7: Hidden feedback loops online bookstore.

- **Prototype Smell**
  As a starting point of developing ML project is to use small-scale prototypes to indicate the behavior of the system. Small-scale prototypes also are used to test multiple scenarios instead of using a large scale system. At the same time, using those small prototypes will not help to build a robust system, in which each small prototype has probably some technical debt and this debt will be increased over time to cover the whole system.

- **Fixed Thresholds in Dynamic Systems**
  Using thresholds in the ML project is very popular, especially in classification and prediction systems. However, selecting the threshold value for the model is usually implemented manually. Therefore, for any new training or testing data, that is required to adjust the threshold again, and this causes debt over time. Figure 2.8 shows the workflow of manipulating the shopping recommendation system offers. The back-end team cares about the model accuracy with delivering the system with a specific confidence level like 50%, whereas the front-end team focuses on how many offers should be presented in the website and they have decided to use a fixed threshold like 70% to decrease the number of presented offers for better user experience. There is no collaboration between the teams and consequently with over time that causes missing the offers which have 50-70% confidence level.

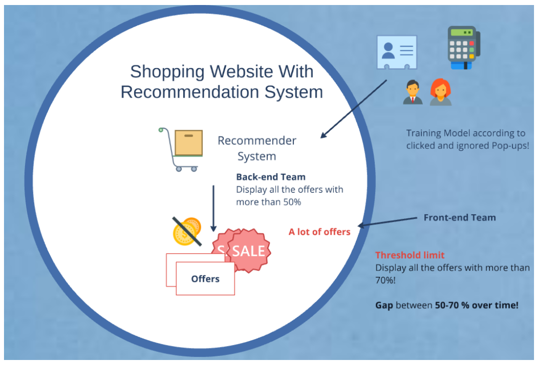

Shopping Website With Recommendation System
Recommender System
Back-end Team
Display all the offers with more than 50%
Training Model according to clicked and ignored Pop-ups!
Front-end Team
A lot of offers
SALE
Offers
Threshold limit
Display all the offers with more than 70%!
Gap between 50-70 % over time!

Figure 2.8: Online bookstore with fixed thresholds in dynamic systems.

- **Up-Stream Producers**
  ML project depends on different up-stream producers, and sometimes some of

these ML models are also consumed by other systems. If there is any failure in the upstream producers, consequently that will affect other models.

- **Reproducibility Debt**
  ML systems depend on the surrounding conditions, so it is required to rerun many experiments for any change in the input, algorithms, and the external world changes to get the required results. This method is called reproducibility, and each rerunning experiment has debt; consequently, that will cause accumulated debt in the long run, especially for real-time ML system.

- **Process Management Debt**
  All the technical debt patterns have discussed the cost of maintaining a single model. However, some models are complicated, and they have hundreds of models that are working at the same time. It is not an easy task for the product manager to determine the models' configurations, the work loading, the potential production value, the pipeline form and the expected user stories in the backlog to implement.

- **Cultural Debt**
  Developing ML software depends on data scientists and software engineers. If this team is a heterogeneous team without taking each one his/her responsibility, that causes a real technical debt. Developing AI products requires multiple engineers from different technical background like ML engineers, software engineers, integration engineers, and so on. The collaboration between those teams is not an easy task, and probably looking for a shared language between them is a necessity in the production process.

Unobserved HTD Pattern

- **Plain-Old-Data Type Smell**
  While implementing ML algorithms, data used (e.g., threshold values, hyper-parameters, input features) and produced are usually encoded into primitive data types such as floating points or integer. In order to make ML model training and testing more effective and efficient, the model should know which data is the threshold setting, or hyper-parameter or output.

3
Case Study Methodology

In this thesis, we are going to conduct a case study following guidelines by Runeson and Höst [8], which will consist of the following elements as proposed by Robson: Objective, the case, theory, research questions, and methods [9]. In the following subsections, we explain all these elements except the theory section which was described in a detailed manner in section 2.

3.1 Objective

The objective of this case study is to investigate HTD patterns, which were originally proposed by Sculley et al. [10], in the prototype models we develop. This effort will assist us to detect and eliminate some of those patterns in the early stage of ML software development lifecycle.
After completing the investigation of the HTD pattern in ML prototypes, we will consider the software system where ML models will be integrated in order to figure out the extent up to which overall system will be affected due to HTD patterns that exist in ML models unless they are removed during productization. This effort will also help us to explore HTD patterns that do not exist in prototypes, but are (likely) to show up after deployment of the ML software system.

3.2 The case

Gothenburg is a quickly growing city in Sweden which suffers from traffic congestion and the empty parking lots problems. Västtrafik has found that it is really valuable to have a helper software tool that advises users where and when they can find suitable empty parking lots during their trips. We plan on using the historical data, which is collected from different parking sensors that will be used as a training dataset to develop ML software.

Prototypes are built for empty parking lots prediction using data obtained from Västtrafik using Microsoft Azure Machine Learning, which is a cloud-based service provided by Microsoft for predictive analytics through machine learning. In the literature, boosted decision tree regression and decision forest regression are the two ML algorithms that proved to be the best performing for empty parking lots prediction [6].

3.2.1 Data Collection

This case study will use a sample dataset which contains the main parking names from a given data collection in the following Table 3.1.
The type of data structure is a tabular dataset, and it contains about 6 million cars in/out events, which were recorded for four years between 2014 and 2017.

Table 3.1: Parking lots data set

Parking ID	Parking Name	Area Name	Municipality	Max Capacity
17	Delsjön-P1	Diseröd	Kungälv	80
18	Delsjön-P2	Diseröd	Kungälv	54
171	Delsjön-P3	Diseröd	Kungälv	72
172	Delsjön-P4	Diseröd	Kungälv	161
87	Lindome station-P1	Maleviksvägen	Kungsbacka	63
88	Lindome station-P2	Maleviksvägen	Kungsbacka	138
174	Lindome station-P3	Maleviksvägen	Kungsbacka	72
112	Radiomotet-P1	Prästängsvägen	Strömstad	134

Each item in this table represents one parking with area name, municipality, and max capacity. Each parking has a collection of historical data events between 2014 - 2017, where the time difference between each consecutive event is 15 minutes.

3.2.2 Technology and Implementation

In this case study, we will use two different ML algorithms, which are boosted decision tree regression, and forest decision regression [6] and develop a system that predicts the empty parking lots. We will use the Microsoft Azure Studio as ML platform for developing ML prototypes [5].
Many corporations have observed the demands for developing ML software. However, developing this type of applications is not a trivial task, and it requires special various skills, tools, and methodologies. For that reason, these companies have decided to simplify the procedure of building ML models by offering libraries (e.g.TensorFlow, etc.), framework and also online cloud solution such as Google, Amazon, IBM, and Microsoft Azure studio. They have offered many facilities like friendly user interface, drag and drop options, importing different types of datasets and other services.

3.3 Research questions

This case study aims to answer the following research question:
RQ1 : Which HTD patterns are observed during early phases of ML software development lifecycle (i.e., prototyping phase)?

3.4 Methods

In this section, we explain how we developed ML prototypes that we later examined for HTD patterns.
Initializing the experiment in Machine Learning Studio is an easy task with drag-and-drop modules pre-programmed with predictive modeling techniques. There are five basic steps to build an ML prototype are explained in the following subsections [14].

3.4.1 Data analysis and preprocessing

This subsection will discuss the following operations regarding data analysis and preprocessing.

Step 1: Get or Select Data
A given data set is a group of tabular data collection with various data which is related to the parking lots. The whole data set is about 6 million records events which cover four years and represent most of the parking lots that are located in the border and the countryside of Gothenburg. But the selected dataset from that given data collection represents around 2 million records.
The reason behind this selection is the importance and completeness of the selected parking lots events regarding Västtrafik recommendation. As a starting point to understand the dataset, we create the ER diagram, which describes the main relations between the data tables. The tabular dataset contains three main columns, which are the parking lot ID, free spaces date, and the number of free spaces.

Step 2: Preparing Data
Parking sensors are the proximity sensors which are tracking and recording the inside and outside movements of the cars in the parking lots. However, parking sensors sometimes suffer from many problems or inaccurate data recorded which cause the main source of the missing and noise data.
Some data is very stable and acquire little variability, while other data swings wildly where the noise data is related to the degree of that swing. Example of this noise data in this case study is the number of free spaces. It has been noticed that the number of free spaces of some parking lots events is more than the maximum parking capacity. In order to solve this problem, it is a good idea to convert the number of free spaces for each event to the percentage where the accepted max value is 100 percent.

Step 3: Feature Engineering
In this step, we explain the features used while building ML models. List of features is shown in Table 3.2.

Table 3.2: List of the potential features

Set	Feature Name	Range	Categorical
A	Weather Temperature	[-15,30]	No
A	Weather Situation	[1,30]	Yes
B	Day Type	[0,1,2,3]	Yes
C	Previous Day % of free spaces	[0,100]	No
D	Previous Week % of free spaces	[0,100]	No
E	Previous 2 Weeks % of free spaces	[0,100]	No
F	Previous 12 Hours % of free spaces	[0,100]	No
G	Day after holiday day	[0,1]	Yes
G	Day before holiday day	[0,1]	Yes
H	Day name for each event	[Monday..Friday]	Yes
H	Time for each event	[00:00 .. 23:59]	Yes

In order to make the features selection effective, we organized the features into feature sets and added each feature set to the elected ML algorithms iteratively. Consequently, we can measure the feature effectiveness separately, adding one feature at a time, and this is related to the evaluation of HTD patterns.
Set A = Weather temperature and situation features
The weather temperature is not a categorical feature since it will be a range of Celsius degrees. In contrast, the weather situation will be categorical feature because there is a certain range of values for this feature (e.g., breezy, windy, snow, foggy, etc.), further details about this categorical feature can be found in Appendix A.1.
Set B = Day Type [Working day - Weekend - National events - School Holidays] (see Appendix A.3).
Set C = Percentage of the free spaces for each event previous day (24 hours).
Set D = Percentage of the free spaces for each event previous week.
Set E = Percentage of the free spaces for each event previous 2 weeks.
Set F = Percentage of the free spaces for each event previous 12 hours.
Set G = Labeled a day which is the after or before the holidays (see Appendix A.2).
Set H = Day name and Time for each event features.

There are different ways to select the most effective features, we will follow the guideline methodology that has been prepared by Azure ML studio to select the most important features. Azure ML Studio employs Sequential Forward Selection (SFS) strategy for feature selection [26].

3.5 Building ML Models

In this case study, we initially developed ML models and later analyzed HTD patterns in these models, referring to the framework by Sculley et al. [10]. In order to build ML models, we use the dataset provided by Västtrafik as we mentioned in section 3.2.1. Dataset consists of the in/out car events log for the parking lots. We divide the dataset into two as the training and the testing datasets. The training dataset will cover three years 2014, 2015 and 2016. The testing data set is just 2017. Features, which are listed in Table 3.2, will be added using Sequential Feature Selection (SFS) technique [26]. ML algorithms that will be employed are Boosted Decision Tree Regression and Decision Forest Regression, respectively. The accuracy and the error percentage in prediction are the main measurements for the models' evaluation.

The development environment is Azure ML studio cloud service where we will follow Azure ML guideline to develop this ML system [23]. This guideline will also help us to perform the feature engineering process and we will try to investigate these HTD patterns throughout this experiment.

3.5.0.1 Implementation Details

The experiments to develop ML models run in stages where each stage represents a part of ML pipeline. Azure ML components will assist to perform each stage aim. This experiment has 11 iterations since there are 11 features which will be added one at a time iteratively according to SFS technique. Figure 3.1 illustrates workflow for building ML models. The first part which displays the different features that we have in this research, the second part represents the used ML algorithms and ML evaluation measurements then the last part is the examination of HTD patterns. As it has been mentioned before, Azure ML studio cloud service is used to implement these experiments. Figure 3.2 displays the Azure ML pipeline, which is the final iteration for ML model development including the total features.

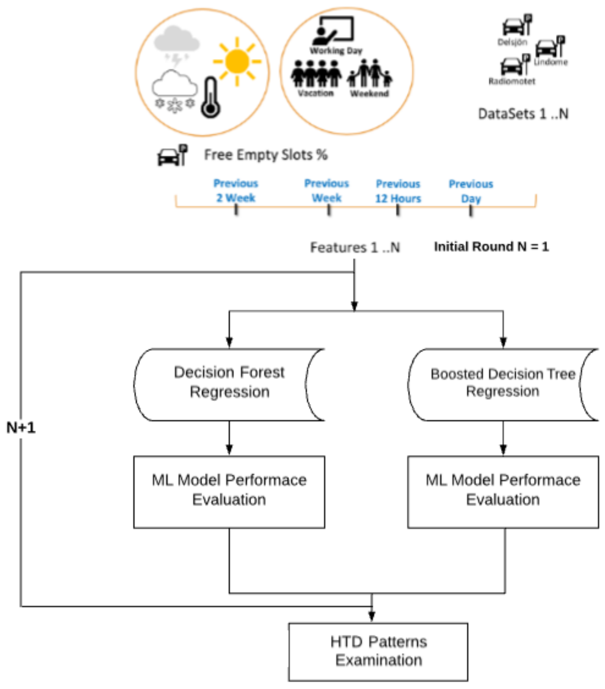


Figure 3.1: ML model development workflow design.

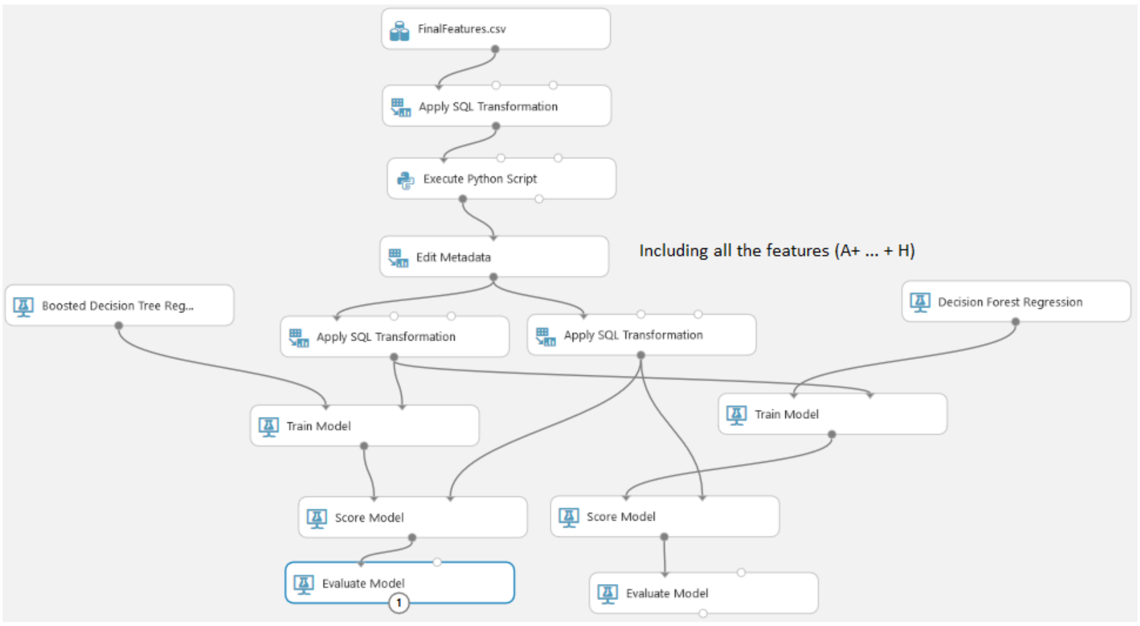


Figure 3.2: Azure ML pipeline implementation for ML models.

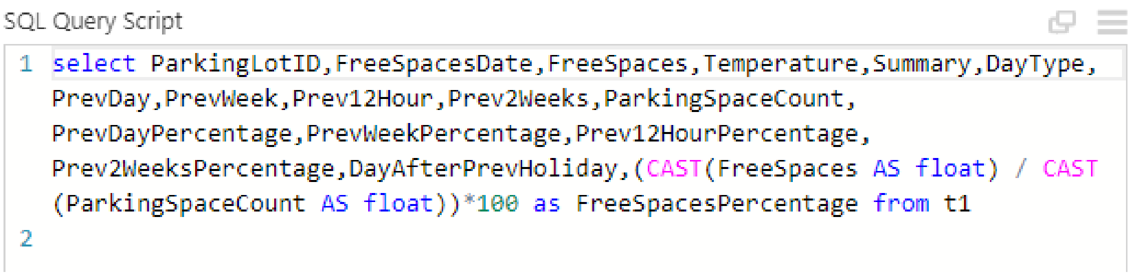
Azure Pipeline Components

As it is has been shown in Figure 3.2, this pipeline contains multiple components, part of are to ML components, and others are with Software Engineering (SE) components. Table 3.3 lists Azure ML pipeline components that were used while building ML models and which group each component belongs to (e.g., ML, SE). We explain each component below.

Table 3.3: Azure components of pipeline implementation for ML components.

Component Name	SE Component	ML Component
Apply SQL Transformation	Yes	No
Edit Metadata	No	Yes
Algorithm Component	No	Yes
Train Model	No	Yes
Score Model	Yes	Yes
Evaluate Model	No	Yes

- **Apply SQL Transformation**
  This component offers the facility to write SQL statements inside Azure ML pipeline. While building ML models, this component is used to prepare training and test datasets. Figure 3.3 illustrates automatically generated query by this component for extraction of features as well as the calculation of some feature values (e.g., empty parking spaces percentage) from the dataset.

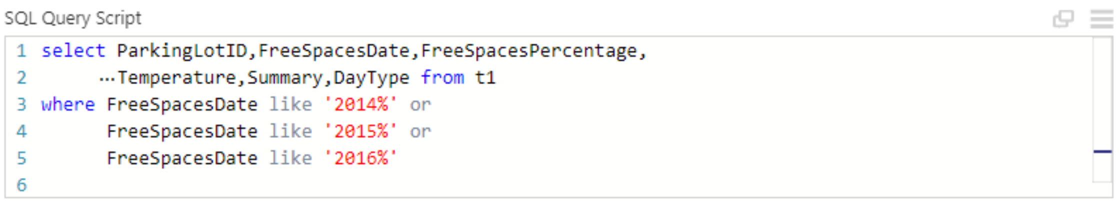


SQL Query Script

```sql
select ParkingLotID,FreeSpacesDate,FreeSpaces,Temperature,Summary,DayType,
PrevDay,PrevWeek,Prev12Hour,Prev2Weeks,ParkingSpaceCount,
PrevDayPercentage,PrevWeekPercentage,Prev12HourPercentage,
Prev2WeeksPercentage,DayAfterPrevHoliday,(CAST(FreeSpaces AS float) / CAST
(ParkingSpaceCount AS float))*100 as FreeSpacesPercentage from t1
```

Figure 3.3: "Apply SQL Transformation" component for preparation dataset.

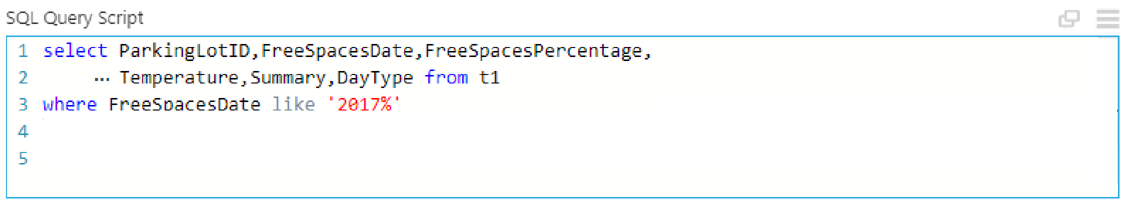
This component is used also to split the dataset regarding the time series into two datasets which are training and test datasets as we can see in Figures 3.4 and 3.5. The training dataset covers three years which are 2014, 2015 and 2016 and the test dataset covers only year 2017.



SQL Query Script

```sql
select ParkingLotID,FreeSpacesDate,FreeSpacesPercentage,
    ...Temperature,Summary,DayType from t1
where FreeSpacesDate like '2014%' or
      FreeSpacesDate like '2015%' or
      FreeSpacesDate like '2016%'
```

Figure 3.4: Example of "Apply SQL Transformation" component to split the data as training dataset.

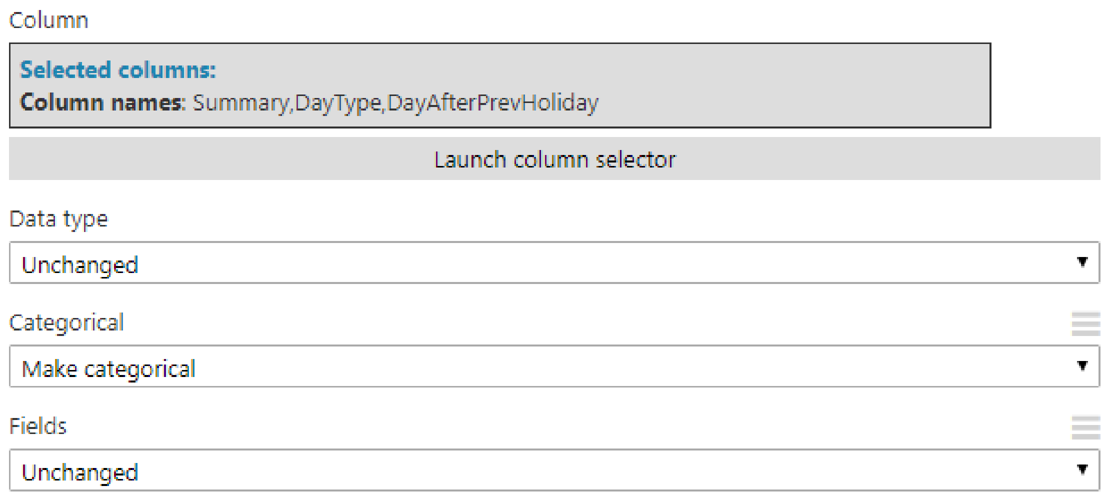


SQL Query Script

```sql
select ParkingLotID,FreeSpacesDate,FreeSpacesPercentage,
    ... Temperature,Summary,DayType from t1
where FreeSpacesDate like '2017%'
```

Figure 3.5: Example of "Apply SQL Transformation" component to split the data as testing dataset.

- **Edit Metadata Component**
  There are two types of features which are categorical and noncategorical. "Edit Metadata" component is used to categorize the features, which can take one value out of a limited group of values (i.e., categorical values). Figure 3.6 displays the categorical features which are weather summary, day type and the day after/previous holiday. We explained previously why these features are considered as categorical features section 3.4.1.



Column

Selected columns:
Column names: Summary,DayType,DayAfterPrevHoliday

Launch column selector

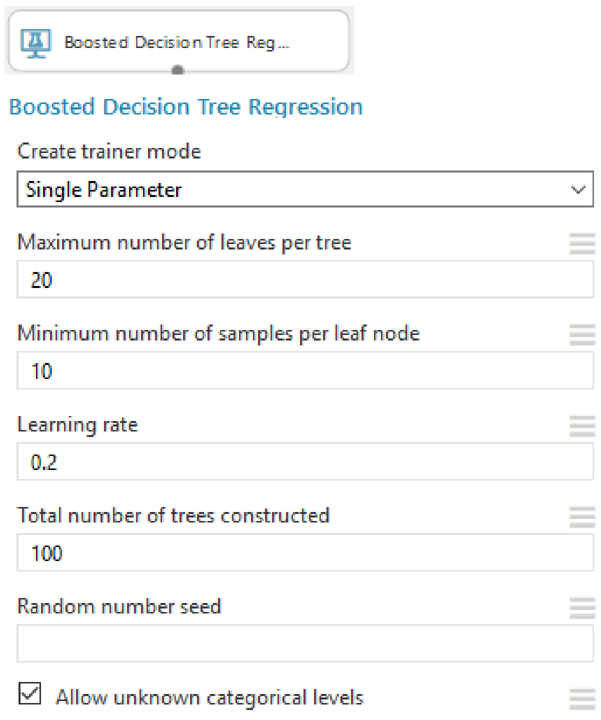
Data type: Unchanged

Categorical: Make categorical

Fields: Unchanged

Figure 3.6: "Edit Metadata" component configuration for preparation dataset.

- **Algorithm Component**
  This component is an encapsulated ML algorithm with the facility to tune many algorithm configurations. In this research, two ML algorithms are used, which are Decision Forest Regression and Boosted Decision Tree Regression. The default Azure ML algorithm component configurations are used in both ML algorithms as we can see in Figures 3.7 and 3.8.



Boosted Decision Tree Reg...

Boosted Decision Tree Regression

Create trainer mode: Single Parameter

Maximum number of leaves per tree: 20

Minimum number of samples per leaf node: 10

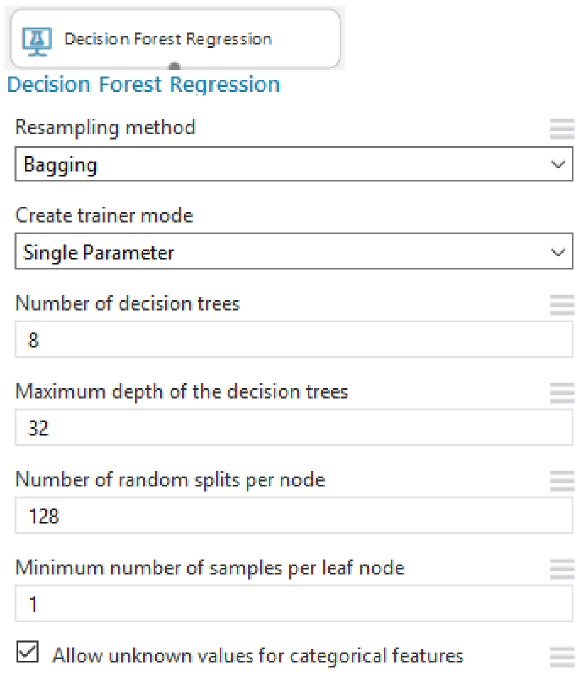
Learning rate: 0.2

Total number of trees constructed: 100

Random number seed:

☑ Allow unknown categorical levels

Figure 3.7: The default configurations of the boosted decision tree regression ML algorithm component.


Decision Forest Regression

Decision Forest Regression

Resampling method: Bagging

Create trainer mode: Single Parameter

Number of decision trees: 8

Maximum depth of the decision trees: 32

Number of random splits per node: 128

Minimum number of samples per leaf node: 1

☑ Allow unknown values for categorical features

Figure 3.8: The default configurations of the decision forest regression ML algorithm component.

- **Train Model**
  This component has two inputs, which are the selected ML algorithm and the training dataset, which is used to learn the model. Decision forest regression and boosted decision tree regression are considered as supervised ML algorithms. In this type of algorithm, it is required to select the target label where this component offers this feature.

- **Score Model**
  This component is responsible for evaluating the predicted values; It is an encapsulated Azure "Blackbox" component.

- **Evaluate Model**
  This component estimates prediction performance values for ML models, which are the coefficient of determination, mean absolute error, and root square relative error.

4
Case Study Results

As we have discussed in Chapter 3, we conducted a case study to examine HTD patterns in the early phase of ML software development lifecycle (i.e., prototyping phase). ML prototypes we previously built for this case study are empty parking lots prediction models, which are planned to be integrated into software system owned by Västtrafik.

Table 4.1 summarizes our case study findings for each HTD pattern. Out of 25 HTD patterns, we were able to detect 12 (i.e., 48%) of them in the ML prototypes and only 1 of these detected HTD patterns was observed only up to some extent (i.e., limited). All the HTD patterns that were detected in the prototypes can propagate to the later stages of ML software development lifecycle and causing problems during productization, testing, or after deployment if precautions are not taken. In addition, 12 HTD patterns (i.e., 48%) that were not detected in ML prototypes could only be detected after deployment of the final ML intensive software. Finally, 1 HTD pattern "Plain Old Data Type Smell" (i.e., 4%) was not detected in this case study due to using libraries rather than implementing ML models from scratch. Moreover, 8 out of 25 HTD patterns (i.e., 32%) are purely due to the nature of ML algorithms and require solutions that are ML specific. While 12 of the HTD patterns (i.e., 48%) require Software Engineering (SE) practices to be detected and removed/managed, remaining 5 HTD patterns (i.e., 20%) require a combination of both ML and SE practices for their detection and removal/management.

In the following subsections, we explain prediction performance results for the ML models we previously built for this case study and then present our findings for the HTD patterns that we detected in those ML models. We also discuss the effect of these HTD patterns in the performance of the overall software system that will result from the integration of ML models into the software owned by Västtrafik. Finally, we discuss HTD patterns that could only be detected after the deployment of the software system.

Table 4.1: HTD patterns during the experiments.

ID	HTD Pattern	Analysis Result		Solution Type	
		Detected in Prototypes	Possible To Detect after Deployment	ML	SE
1	**Entanglement**	X	-	X	-
2	***Correction Cascades***	-	X	-	X
3	***Undeclared Consumers***	-	X	-	X
4	***Unstable Data Dependencies***	-	X	-	X
5	***Legacy Features***	-	X	X	-
6	**Bundled Features**	X	-	X	-
7	**ϵ-Features**	X	-	X	-
8	**Correlated Features**	X	-	X	-
9	***Direct Feedback Loops***	-	X	X	-
10	***Hidden Feedback Loops***	-	X	-	X
11	**Glue Code**	X	-	-	X
12	**Pipeline Jungles**	X	-	-	X
13	**Dead Experimental Code Paths**	X	-	-	X
14	**Abstraction Debt**	X	-	X	X
15	Plain Old Data Type Smell	-	-	-	X
16	**Multiple Language Smell**	X	-	-	X
17	***Prototype Smell***	-	X	-	X
18	**Configuration Debt**	X	-	X	X
19	***Thresholds in Dynamic Systems***	-	X	X	X
20	**Prediction Bias**	X	-	X	-
21	***Up-Stream Producers***	-	X	-	X
22	**Data Testing Debt**	X	-	X	-
23	***Reproducibility Debt***	-	X	X	X
24	***Process Management Debt***	-	X	-	X
25	***Cultural Debt***	Limited	X	X	X

4.1 Results for Developed ML Models

In this section, we will discuss the results where it contains 11 iterations as has been described in section 3.5. The coefficient of determination, which represents the model accuracy performance, will be used as a comparison criterion in the obtained results.

Table 4.2 and Figure 4.1 illustrate the change of the coefficient of determination and the error percentage as a result of using Sequential Forward Selection (SFS) feature selection technique [26] in order to add 11 features iteratively. Boosted decision tree regression is used as ML algorithm. Obviously, feature A does not have any impact on model accuracy, which is -0.09. When the feature B was included, the model accuracy increased to 0.038. Feature B affected the model, although the model accuracy percentage is not that good. Now, the model accuracy was increased dramatically up to 0.68 by adding feature C. After that, the model accuracy was keeping up to reach 0.78 by adding the feature D where this feature also has a significant effect on the model accuracy. The model accuracy was increased steadily by adding these features, which are E, F, and G. However, those features did not have that obvious impact on the model. Probably, the last feature, which is H, had a good effect on the model in which the model accuracy reached 0.9.

Table 4.2: The result of adding features iteratively by using boosted decision tree regression algorithm.

Features	MAE	RMSE	Coefficient of Determination
A	27.8	37.6	-0.09
A+B	25.5	35.2	0.038
A+B+C	11.7	20.2	0.68
A+B+C+D	9.67	16.5	0.78
A+B+C+D+E	9.89	16.4	0.79
A+B+C+D+E+F	9.4	15.9	0.80
A+B+C+D+E+F+G	8.89	15.2	0.82
A+B+C+D+E+F+G+H	6.38	11.38	0.9

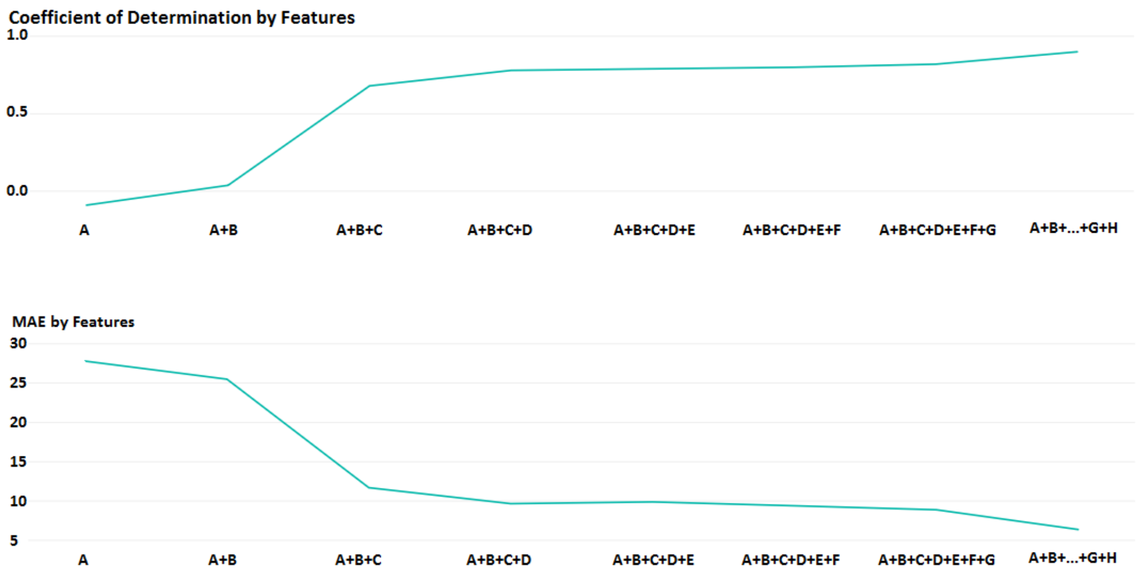


Figure 4.1: The correlation between the features, mean absolute error and coefficient of determination with boosted decision tree regression algorithm.

Table 4.3 and Figure 4.2 display the change of the coefficient of determination and the error percentage for the decision forest regression algorithm as a result of adding the 11 features according to SFS feature selection technique [26]. Clearly, feature A had a bad impact on the model accuracy which is -0.27. When the feature B was inserted, the model accuracy increased to -0.15. Feature B is not considered as an effective feature but it enhanced the model accuracy anyway. Now, the model accuracy increased substantially to 0.55 after feature C was added. After that, the model accuracy remained at 0.77 after feature D was added where this feature also had a significant effect on the model accuracy. The model accuracy was enhanced a little bit by adding feature E. Feature F was not affected the model in a good way. But the model accuracy is still acceptable. The last two features, G and H had a good effect on the model in which the model accuracy reached 0.88.

Table 4.3: The result of adding features iteratively by using decision forest regression algorithm.

Features	MAE	RMSE	Coefficient of Determination
A	27.9	40.57	-0.27
A+B	26.6	38.6	-0.15
A+B+C	15.1	24.2	0.55
A+B+C+D	10.45	17.27	0.77
A+B+C+D+E	9.7	16.21	0.8
A+B+C+D+E+F	10.35	17.17	0.77
A+B+C+D+E+F+G	9.23	15.6	0.81
A+B+C+D+E+F+G+H	6.63	12.05	0.88

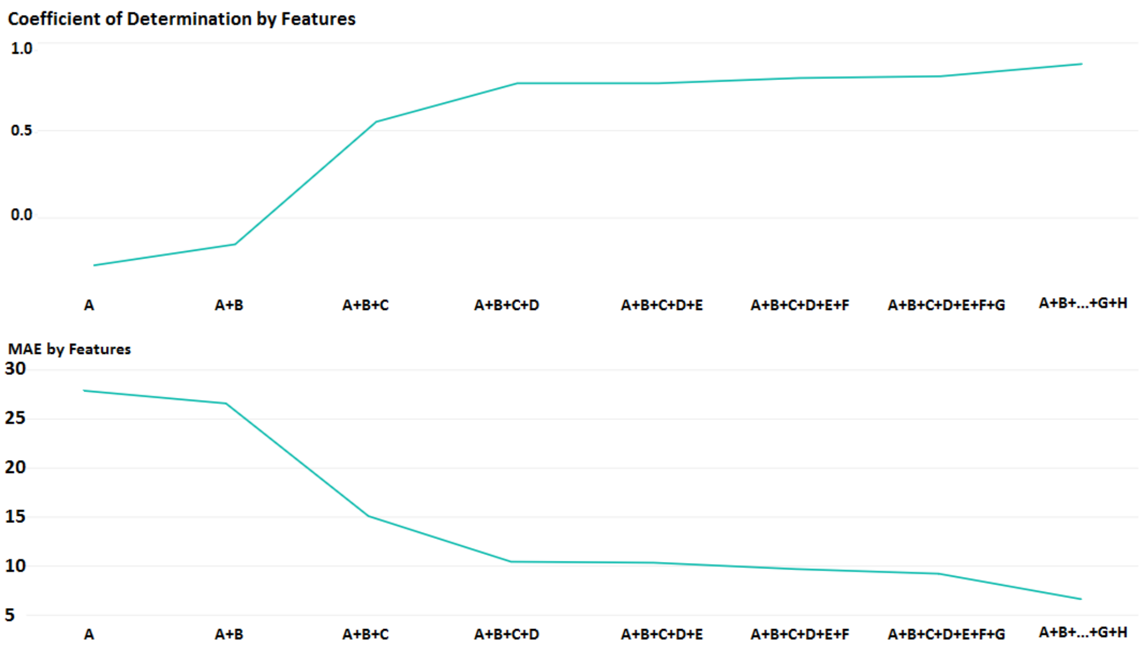


Figure 4.2: The correlation between the features, mean absolute error and coefficient of determination with decision forest regression algorithm.

The changes in the measurements of both algorithms were very close. However, the percentage results of the boosted decision tree regression were higher than the forest one by 0.02 percentage points.

As a cross-validation for the obtained results, we conducted an experiment and the used features for this experiment were B, C and D. Table 4.4 and 4.5 illustrate the obtained results for both ML algorithms which are boosted decision tree regression and decision forest regression. Obviously, that B, C, and D features are quite enough to build a mature model with respecting to the two machine learning algorithms.

Table 4.4: The result of adding B, C and D features by using decision forest regression algorithm.

Features	**MAE**	**RMSE**	**Coefficient of Determination**
B+C+D	7.9	14.4	0.84

Table 4.5: The result of adding B, C and D features by using boosted decision tree regression algorithm.

Features	**MAE**	**RMSE**	**Coefficient of Determination**
B+C+D	7.4	13.08	0.88

4.1.1 Observed HTD Patterns

In this section, we explain HTD patterns that are discovered in the ML models we developed, together with the data analysis techniques we employed to discover those HTD patterns.

1. Entanglement
"CASE: Changing Anything Changes Everything"
Sculley et al. [10] mentioned some examples like the effects of adding and removing features, changing learning settings, using hyper-parameters and data selection [10]. We observed that adding or removing features causes changes in the resulting models' prediction performance. Tables 4.4 and 4.5 illustrate significant effects of adding features B, C and D. Also, removing some features like feature A enhanced the model accuracy as well.
Entanglement is an ML specific issue and as mentioned previously in section 2.2.2, one can employ an ML specific solution such as using ensemble models [7] to make the system robust against entanglement. There are hybrid solutions, which require both ML and SE knowledge, such as implementation and usage of visualization tools to tackle changes and isolation of models.

2. Bundled features and ϵ-Features
Sculley et al. [10] mentioned that sometimes a group of features is added to the

model according to the prior expectations or to make the work done quickly. In this experiment, we tried to figure out the importance of each feature independently by adding the features iteratively. As it has been shown in Figures 4.1 and 4.2, adding features like B, C, and D had a significant effect on the coefficient of determination value or the accuracy performance. It is also clear that building a model using only B, C, and D as input features would be enough and adding the remaining features is complicating the model without any significant contribution to the model performance. However, the feature bundle consisting of E, F, G, and H had no significant effect on the accuracy performance value or the error percentages. Hence, E, F, G, and H are considered **bundled features**. Moreover, feature A, and each of the features in the bundled features "E, F, G and H" are individually ϵ-features. Detection of bundled and ϵ-features require ML-base solutions, such as employing feature selection techniques.

3. Correlated features
Sculley et al. [10] defined the correlated features as a pair of features that are correlated but only one feature of that pair has a significant effect on the models' performance. They also stated that it is not easy to detect this type of correlation. For that reason, we employed ML/statistics based approach and created the correlated features map over time for all used features. In this case, we are able to analyze then detect correlated features.

Figures 4.3 and 4.4 show the correlations among the features for the first quarter (i.e., “Q1” for January, February, and March), second quarter (i.e., “Q2” for April, May, and June), third quarter (i.e., “Q3” for July, August and September) and fourth quarter (i.e., “Q4” for October, November, and December) of the years 2014 and 2015, respectively.
In Figure 4.3, the correlation between features C and E for “Q1” in 2014 is 0.50 whereas the correlation increases gradually in “Q2” to 0.75 then this correlation drops down to 0.6 in “Q3”. In “Q4”, this correlation rises dramatically to 0.9. Regarding the previous experiments, feature E does not have any impact on the model performance, and it is considered as a ϵ-feature. However, this strong correlation may affect the model performance negatively. The correlation between features C and A for “Q1” in 2014 is 0.45 whereas the correlation increases gradually in “Q2” to 0.75 then this correlation drops down to 0.45 in “Q3”. In “Q4”, this correlation rises to 0.9. Feature A also does not have a significant effect on the model performance, and it is considered as a ϵ-feature.

In Figure 4.4, the correlation between the features C and E are 0.70 and 0.75 during the quarters “Q1” and “Q2”, respectively. However, the correlation between these two features later drops down to 0.20 and 0.30 during the quarters “Q3” and “Q4”, respectively. As we can see in Figure 4.4, while the feature C has a significant impact on the prediction performance, the contribution of feature E is quite insignificant. During the quarters “Q1” and “Q2”, the feature E will be highly correlated to the predicted output, due to its high correlation with the feature C. Just looking at these values might give a wrong idea that this feature contributes to the prediction performance since such high correlation would be due to the high correlation between this feature and another feature that actually has a significant contribution in the prediction performance of the ML algorithm. However, as shown in Figure 4.4 correlation between this feature pair is quite brittle and they are correlated features according to the HTD framework by Sculley et al. [10]. If one decides to prefer adding feature E (i.e, “% of empty parking lots for previous 2 weeks”) into the feature set instead of the feature C (i.e, “% of empty parking lots for previous day”) based on his observations of the quarters “Q1” and “Q2”, the prediction performance will get worse for the quarters “Q3” and “Q4”. As it can be seen in Figure 4.4, the features C (i.e, "% of empty parking lots for previous day”) and D (i.e, “% of empty parking lots for the previous week”) are also correlated features.

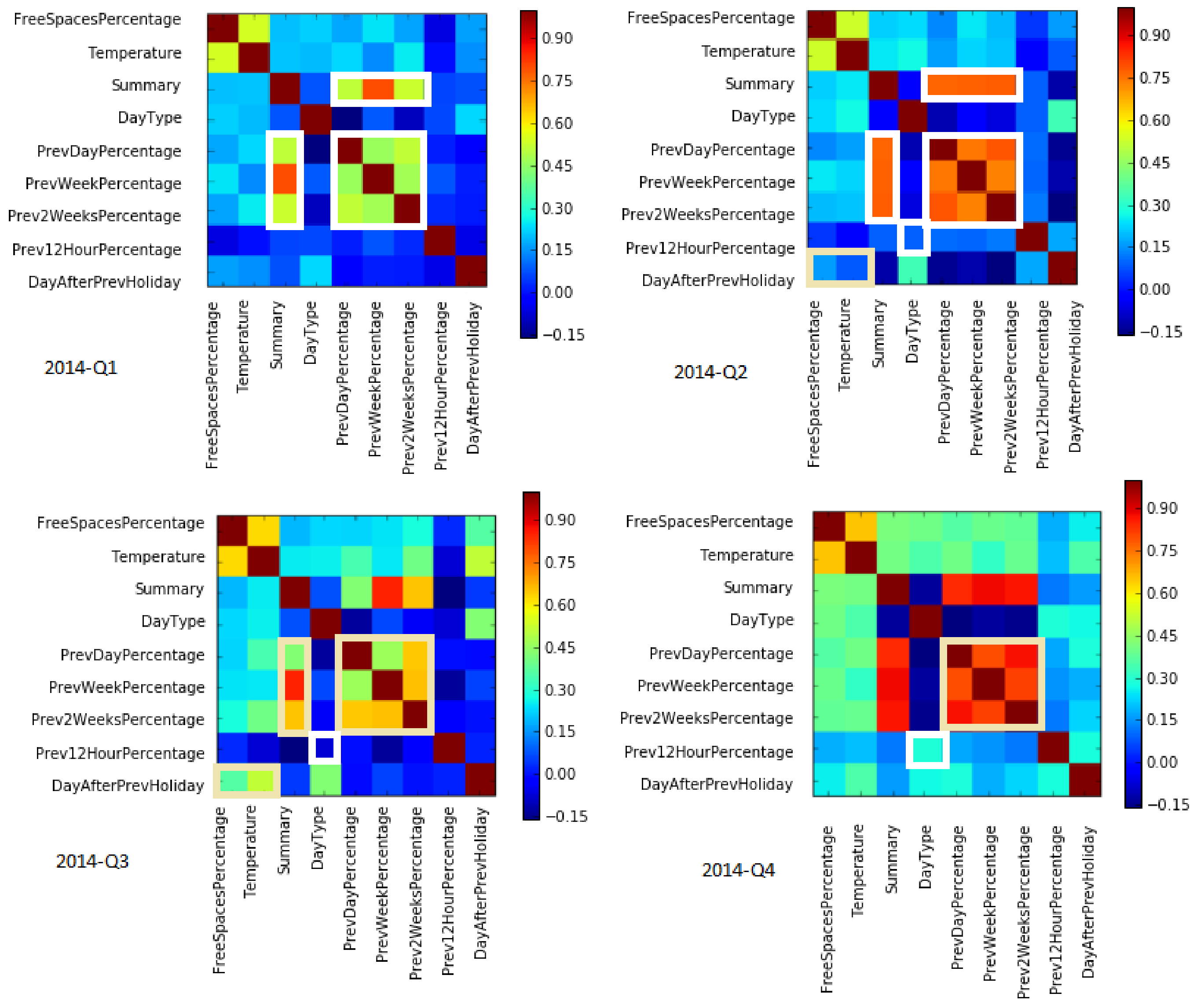


Figure 4.3: Features correlation map for Delsjön parking the four quarters of the year 2014.

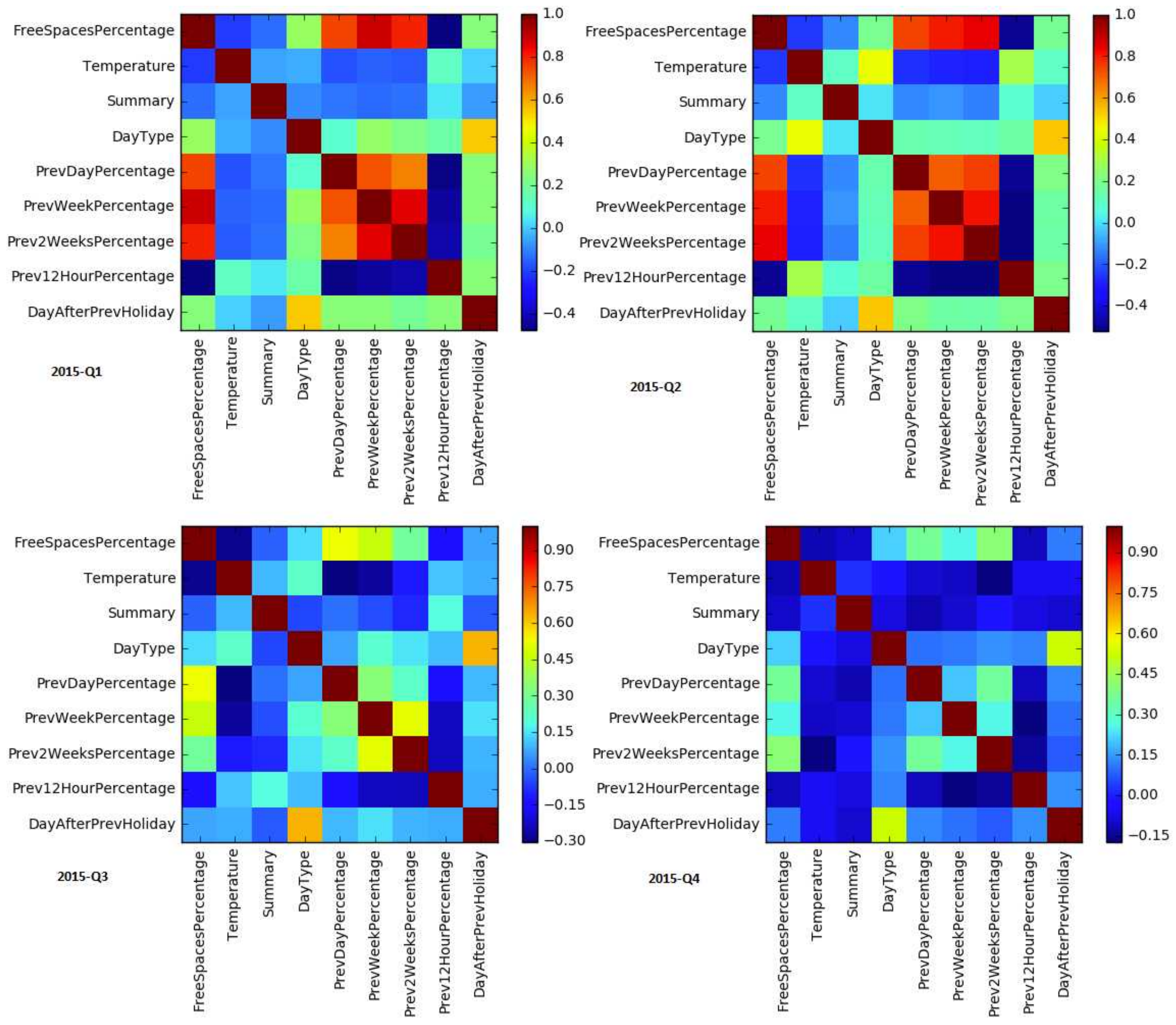


Figure 4.4: Features correlation map for Delsjön parking the four quarters of the year 2015.

4. Glue code

There are many reasons that cause the glue code pattern, one of them is using the generic packages regarding as indicated by Sculley et a. [10]. They have mentioned that the resulting ML software might contain 5% ML code and 95% glue code [10]. The glue code, which is an SE-related problem, is generated in different stages of the ML pipeline and especially in the data preparation stage. Also, we can generate this pattern obviously when using the black-box cloud online service. Sculley et al. [10] mentioned some SE practices to prevent/remove this pattern like wrapping the black-box into APIs to decrease the cost of adjusting the packages. While building ML models, data processing was required to prepare the training and test dataset for the model as input. This process contains many components that are considered glue code as we can see in Figure 4.5, where components, such as "Split Component" (see Figures 3.4 and 3.5) besides "Apply SQL Transformation" (see Figure 3.3), were used to prepare ML models. In this experiment, we could not follow exactly Sculley et al. [10] framework to wrap the black-box since we are using Azure ML cloud service. Clearly, all the used components were not ML but SE components. Also, Azure cloud offers the ability to write, refactor and optimize the code inside the components. Since the components that cause this HTD pattern were detected during the early stage of development, it is easy to apply one of the following solutions. One of the suggested solutions is to apply the traditional software practice like code refactoring and reducing the number of the used programming languages. Another solution, that is recommended by Sculley et al. [10], is using APIs instead of consuming the data from static components. In such case, the pipeline would include components that communicate with these APIs instead.

5. Pipeline jungles
Sculley et al. [10] have defined the pipeline jungles as a certain case of glue code, and this pattern is related to the input data for the ML model. They stated that to avoid this pattern, it is required to take a comprehensive look for the data collection and the feature engineering. We can notice the pipeline in Figure 4.5 besides SQL code in the following Figures 3.3, 3.4 and 3.5. The output training and testing dataset are tabular raw data with numerical values. Raw data is transformed into a numerical representation that ML algorithm can understand. As we suggested before, employing SE practices such as using APIs instead of the static components is still a good solution. Sculley et al. [10] also suggested "hybrid research" team to implement the ML system where the engineers and the researchers work together to build the ML system architecture and reduce this cost. In this experiment, the ML system prototype was implemented by a software engineer, and it was supervised by ML researcher industrial supervisor. Now, a proof of concept for this ML system was developed, and after this pattern was detected in this stage, it will be in the consideration during the production stage of this ML intensive system development.

6. Dead experimental code paths
Sculley et al. [10] indicate that this pattern, which is a SE related issue, appears in ML models since there are glue code and pipeline jungles patterns in the resulting models. They mentioned that implementing multiple experiments with many paths might cause many problems. For example, if the model has never used a certain path and the path has never been tested before. Then in the production stage, the experimental path might be pushed back to the main branch in the code versioning system. Sculley et al. [10] suggested reexamining each experiment path to remove the unused paths as SE oriented solution. In this experiment and as we have mentioned before, we used Azure ML guideline to implement ML models [23], and we added the features cumulatively as we can see in Appendix A.4. It was clear that there were many experiments paths, but since there were a limited number of features, after the bundle, ϵ-features and the correlated features were analyzed. Consequently, the dead experiments paths were detected during this case study.

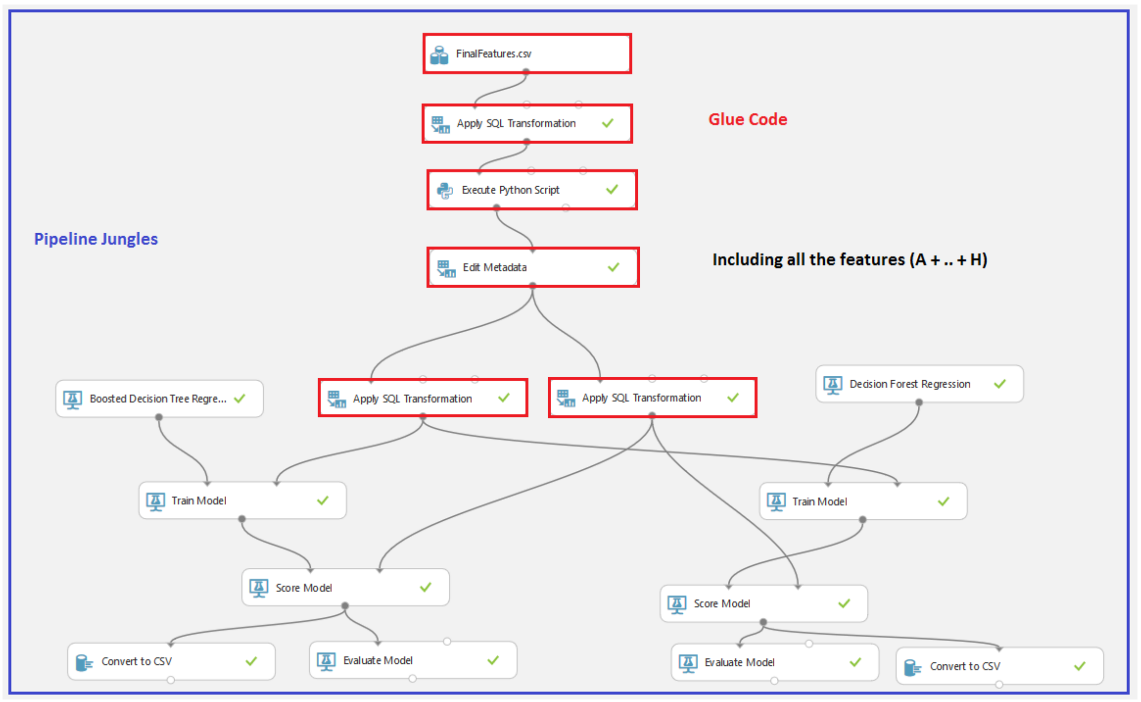


Figure 4.5: Glue code and pipeline jungles in the experiments.

7. Abstraction debt

Abstraction Debt as it was indicated by Zheng [28] and Sculley et al. [10], unlike relational database as an abstraction for many database management systems, there is no distinct abstraction to support ML systems. Sculley et al. [10] indicate that there is no abstraction to describe a stream of data, model or prediction, and such deficiency becomes much more obvious in distributed ML learning. Like any other ML application, our ML models suffer from abstraction debt. However, although our ML implementations can be regarded as distributed learning due to using Microsoft Azure ML, which is a cloud application, due to the black-box nature of the technology we are using, abstraction debt was observed up to a limited extent in our ML prototypes.

8. Multiple-language smell

Sculley et al. [10] discussed that using multiple programming languages will increase the cost of testing. While building ML models, three programming languages are used, which are SQL, Python, and encapsulated ML code in MS Azure on the cloud consisting of modules implemented in R and Python. Later, as we will see in section 4.2, many programming languages could be used to deal with the ML model. This pattern is considered as a traditional SE issue. The careful tracking for the code updates is a suggested solution to detect this pattern. Employing an SE oriented solution such as using unit testing for each code portion of the ML pipeline system helped a lot to eliminate this pattern after its detection during the case study. We have tested separately the SQL code in PostgreSQL then we used this code inside the Azure ML pipeline system, and we did the same for the Python code by using a special Python development environment.

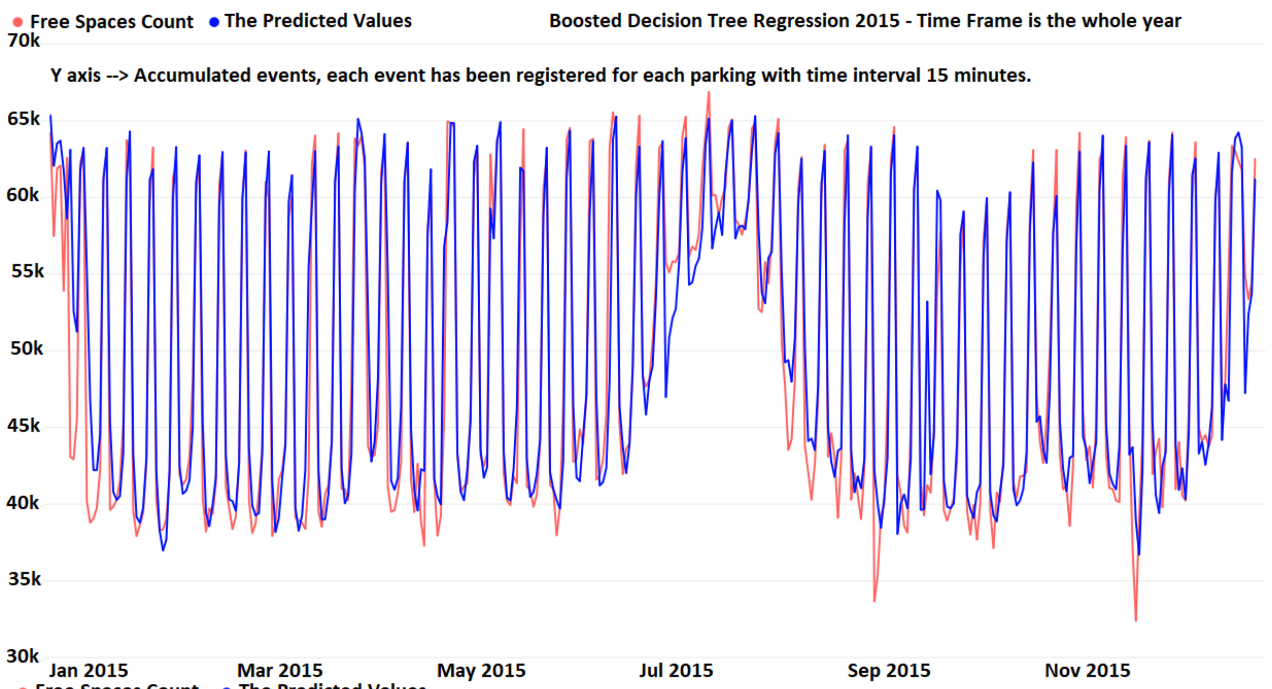
9. Configuration debt
Sculley et al. [10] stated that "any large system has a wide range of configurable options, including which features are used, how data is selected, a wide variety of algorithm-specific learning settings, potential pre- or post-processing, verification methods, etc." [10]. In the ML models built, the selected features and the learning settings of the used ML algorithms were the observed configurations debt. Early in this section, we explained the bundled, ϵ-features, and the correlated features as HTD patterns. Moreover, we discussed the impact of these patterns and how we could detect them. Investigation of all these features related to HTD patterns helps elimination of the configuration debt. Boosted decision tree regression and decision forest regression were used in this experiment. Tables 4.2 and 4.3 illustrate the obtained results of the two ML algorithms and how some features were affected those ML algorithms in a different way. For instance, when the boosted decision tree regression algorithm was used, and the feature F was added, the model accuracy was increased by 0.01 percentage points. In contrast, when the decision forest regression algorithm was used and the feature F was added, the model accuracy was decreased by 0.03 percentage points. Increase in the number of features used might result in configuration debt, since each feature might introduce complexity into the configuration details for the software. For instance, a feature might not be available before a certain date, codes that preprocess features might differ because of the changes in the logging format, a feature used in prototyping may not be available for production. All such information should be stored in configuration files. SE practices such as code review of configuration files should be employed as well as ML approaches to eliminate features (e.g., feature selection techniques) that do not have significant effects on the model performance. In this way, we can prevent the increase in the number of features and hence reduce the complexity of configuration files.

10. Prediction bias
In order to monitor and test the ML models development process, we run an extra experiment with 3 iterations:

Iteration 1: Training dataset covers one year which is 2014, and the testing dataset covers the year 2015. Table 4.6 illustrates the obtained results for the selected training and testing data where the two ML algorithms were used. Figures 4.6 and 4.7 illustrate the distribution of the predicted and the actual data for the year 2015 and January 2015, respectively.

Table 4.6: Results of ML algorithms which trained with all features for the whole data set of 2014 and tested by the year 2015 of historical parking events.

ML Algorithms	COF[Accuracy]	MAE	RMSE
Boosted Decision Tree Regression	0.85	6.95	12.55
Decision Forest Regression	0.84	7.29	13.26



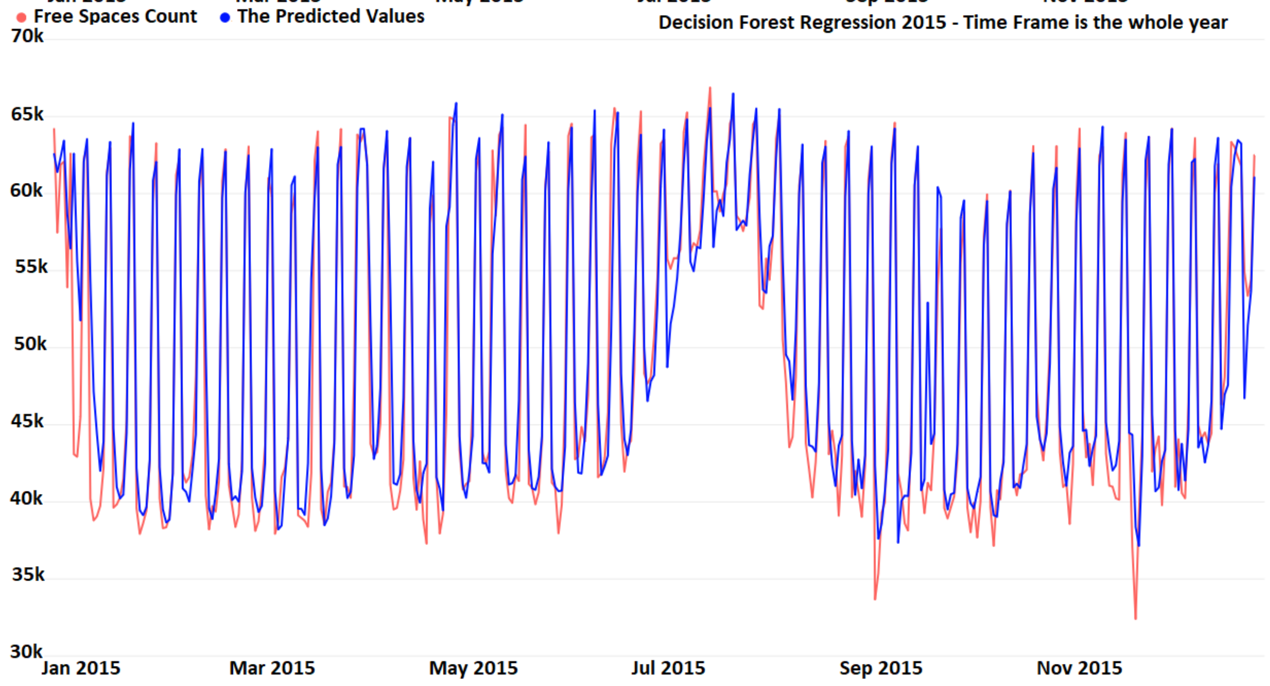
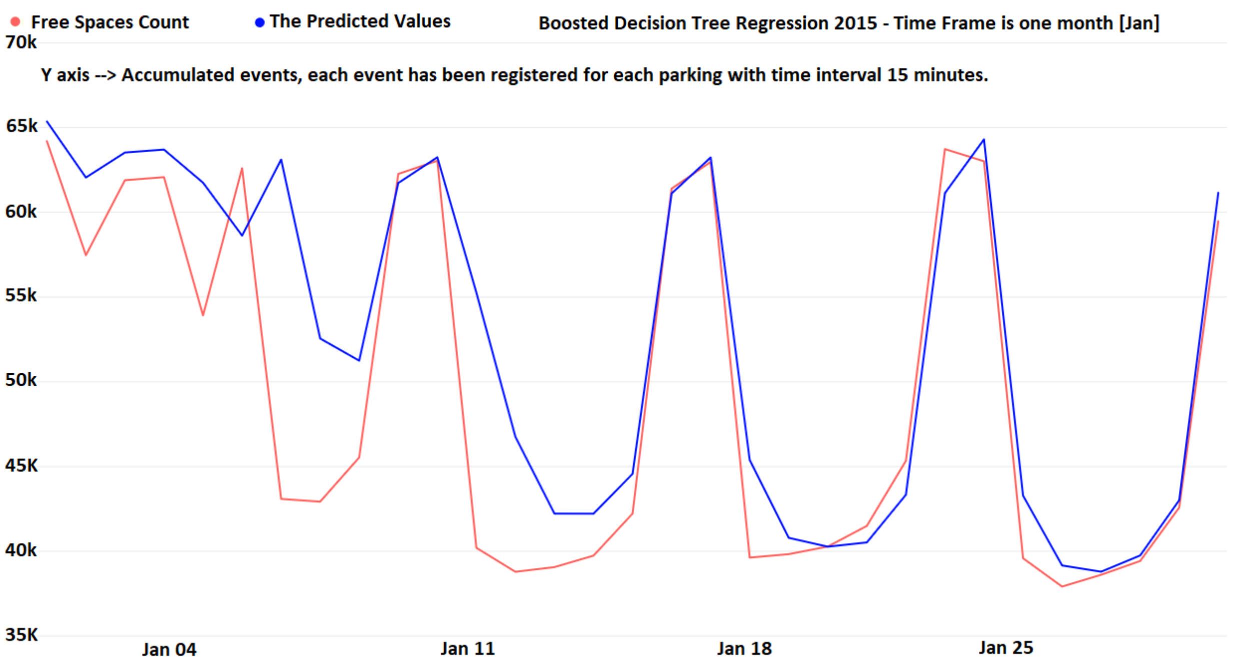


Figure 4.6: The distribution of the actual and the predicted dataset for one year.



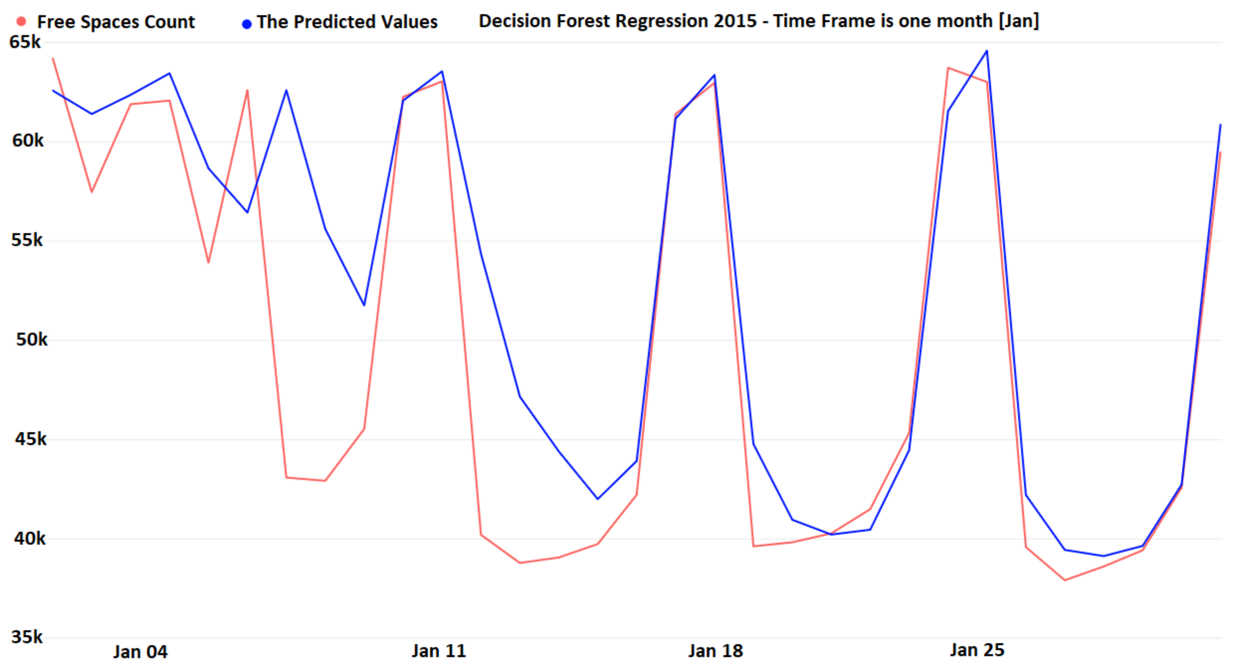


Figure 4.7: The distribution of the actual and the predicted dataset for one month.

Iteration 2: Training dataset covers two years which are 2014 and 2015, and the testing dataset covers the year 2016. Table 4.7 illustrates the obtained results for the selected training and testing data where the two ML algorithms were used. Figures 4.8 and 4.9 display the distribution of the predicted and the actual data for the year 2016 and January 2016, respectively.

Table 4.7: Results of ML algorithms which trained with all features for the whole data set of 2014 2015 and tested by the year 2016 of historical parking events.

ML Algorithms	COF[Accuracy]	MAE	RMSE
Boosted Decision Tree Regression	0.89	6.09	10.48
Decision Forest Regression	0.89	6.26	10.95

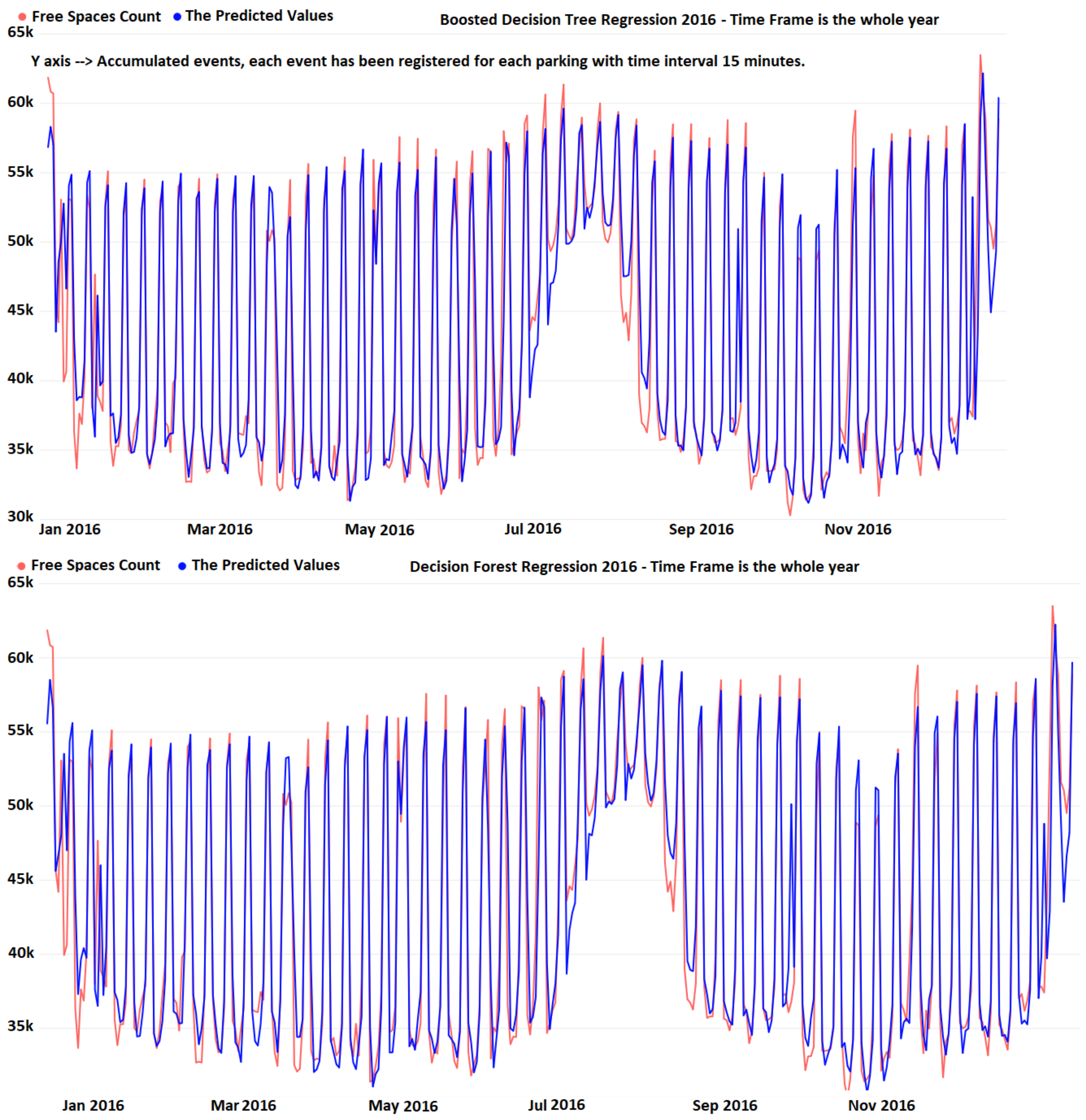


Figure 4.8: The distribution of the actual and the predicted dataset for one year.

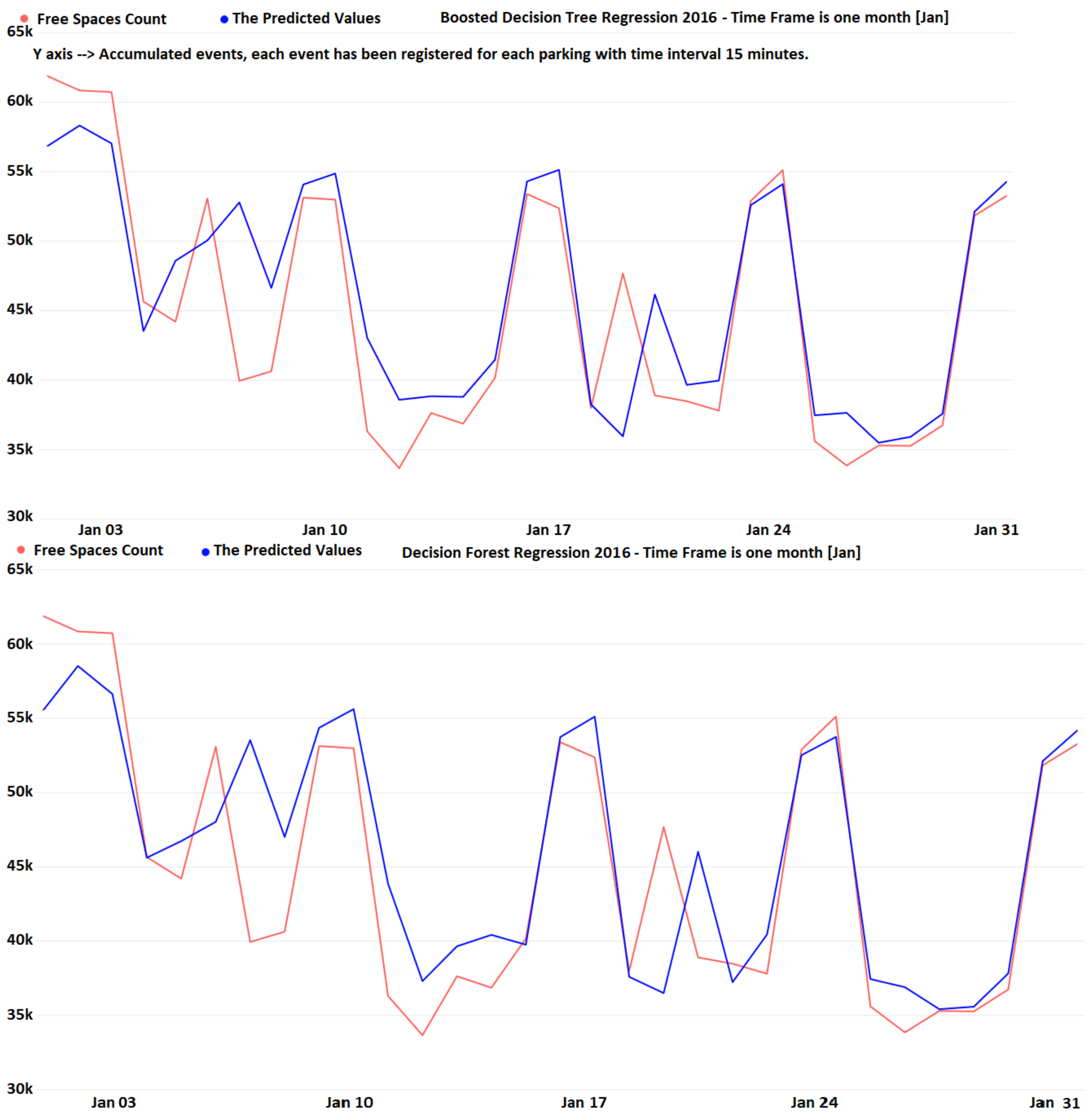
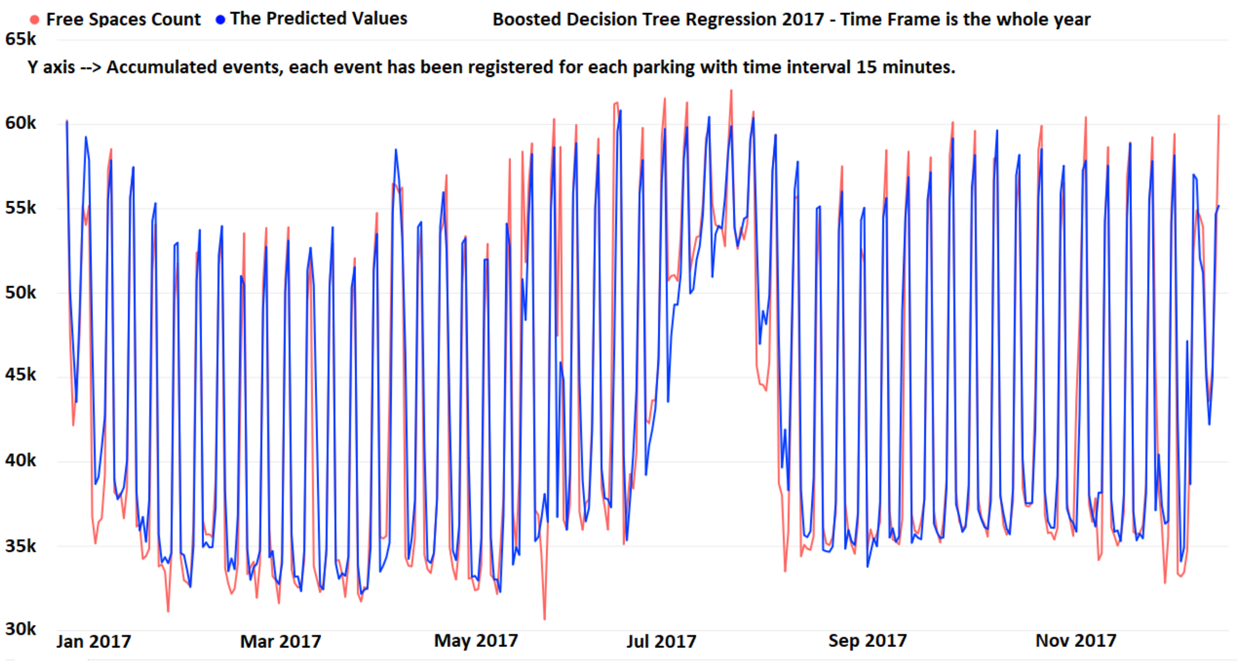


Figure 4.9: The distribution of the actual and the predicted dataset for one month.

Iteration 3: Training dataset covers three years which are 2014, 2015, 2016 and the testing dataset covers the year 2017. Table 4.8 displays the obtained results for the selected training and testing data where the two ML algorithms were used. Figures 4.10 and 4.11 display the distribution of the predicted and the actual data for the year 2017 and January 2017, respectively.

Table 4.8: Results of ML algorithms which trained with all features for the whole data set of 2014, 2015, 2016 and tested by the year 2017 of historical parking events.

ML Algorithms	COF[Accuracy]	MAE	RMSE
Boosted Decision Tree Regression	0.90	6.36	11.25
Decision Forest Regression	0.88	6.63	12.05



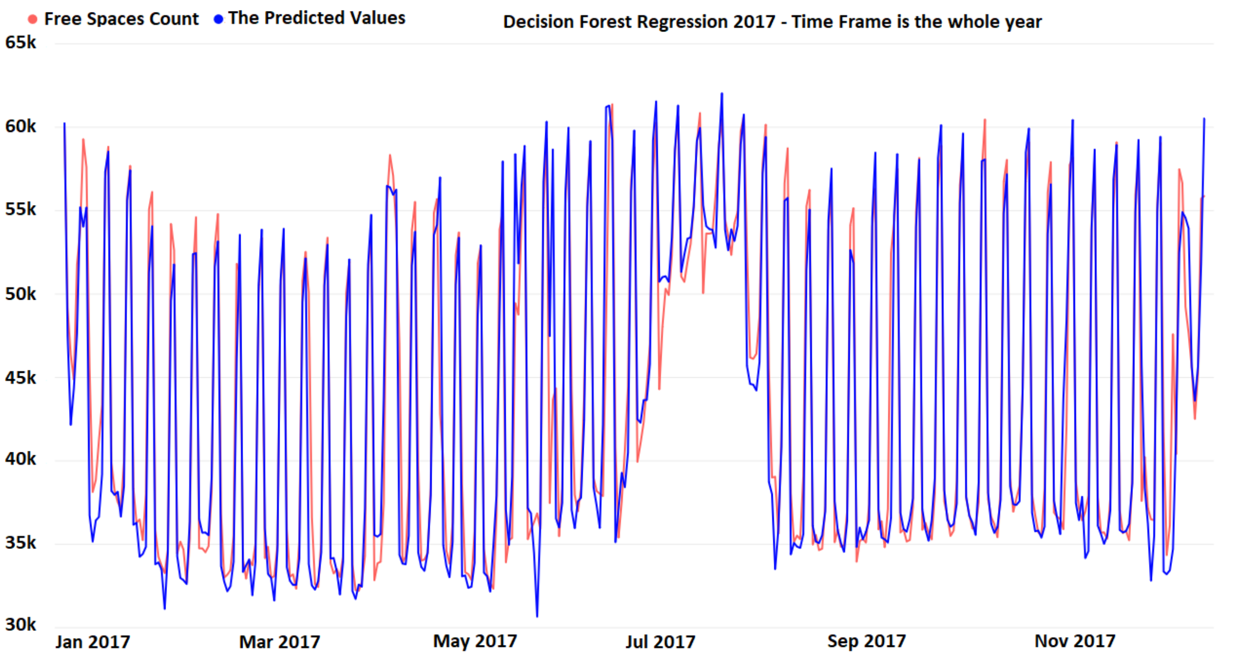
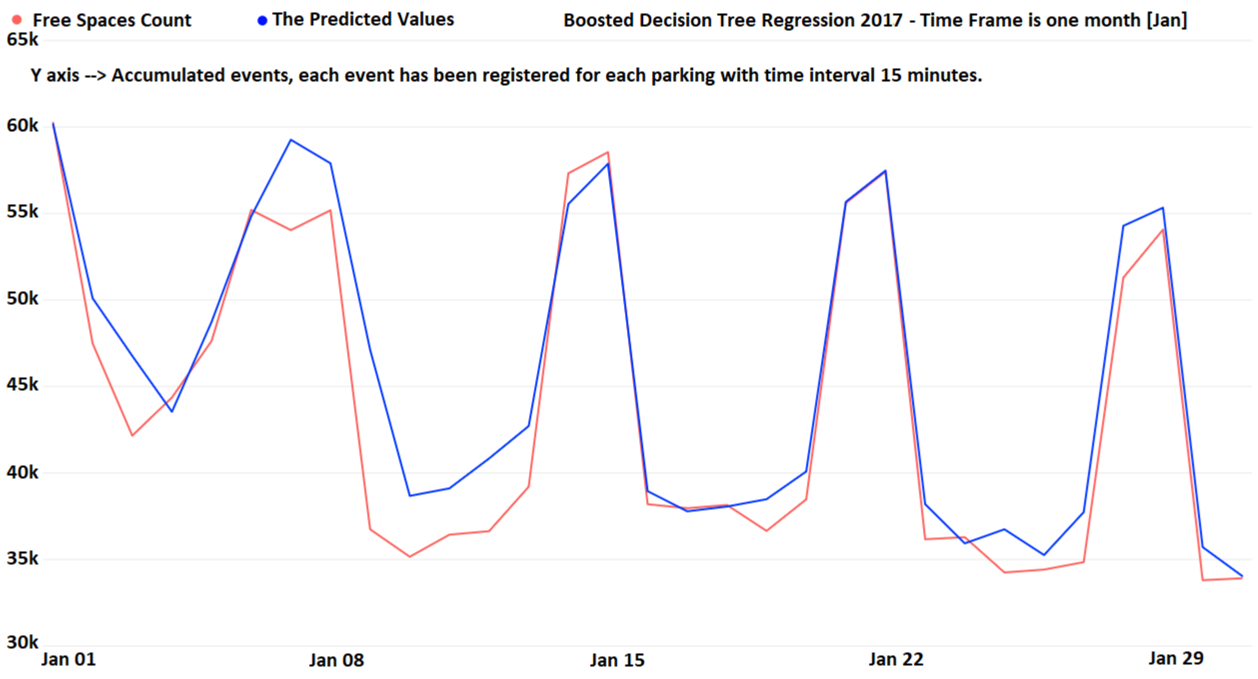


Figure 4.10: The distribution of the actual and the predicted dataset for one year.



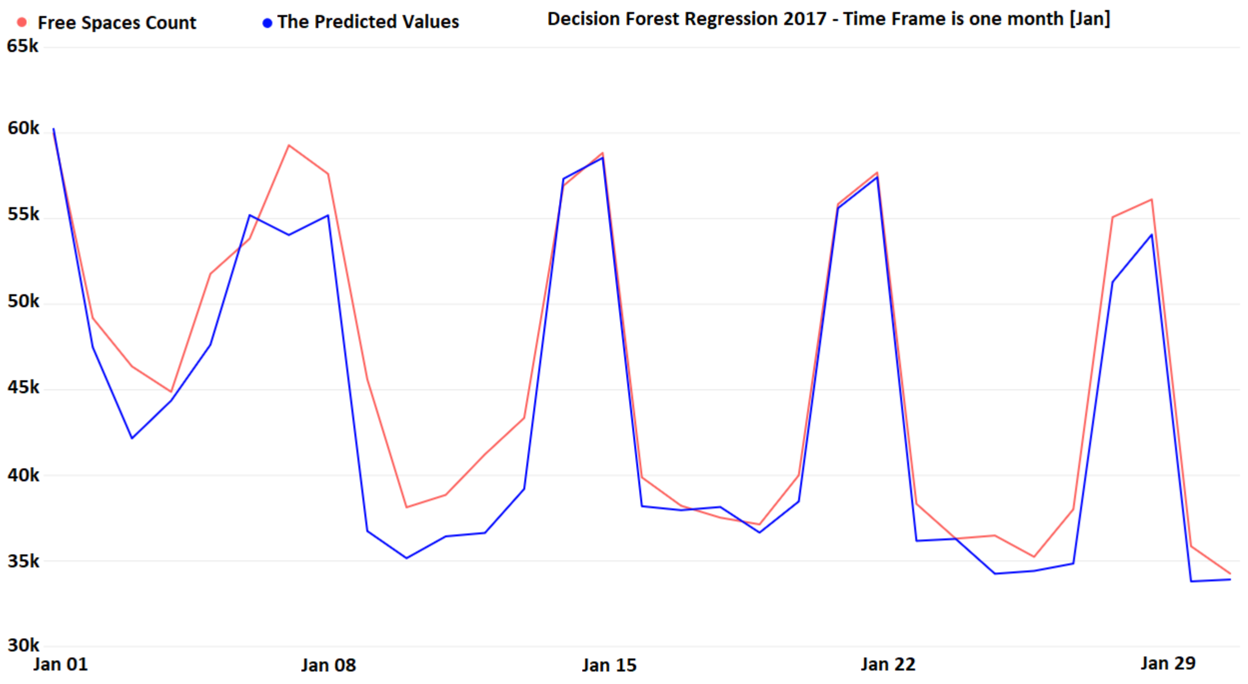


Figure 4.11: The distribution of the actual and the predicted dataset for one month.

In Figures 4.7, 4.9 and 4.11, we can notice the differences between the distribution of the predicted and the actual data for the ML model by using the different data size for each iteration. The difference between the predicted, which is the blue line, and the actual data, which is the red line is high in Figure 4.7 for iteration 1. In iteration 2, the difference between the blue line and the red one is less than iteration 1 as we can see in Figure 4.9. The two lines are almost identical for the last iteration 3 as we can see in Figure 4.11. We can say that the model accuracy for the last iteration is the best. Also, the predicted and the actual data distribution for iteration 3 are also very close.

Sculley et al. [10] state that prediction bias is revealed due to the deviation of distribution of the predicted values from the actual values. Such deviations are due to the fact that data behavior changes and hence ML models built may not perform as expected while making inferences using the most recent data. One technique Sculley et al. [10] proposed to tackle prediction bias is slicing the dataset and testing how each data portion affects the model. Another example is using the most recent data to train ML models instead of the old data to avoid any surprising changes in the data behavior. In order to investigate prediction bias in the ML models that we built, we followed Sculley et al. [10] framework and we sliced the dataset according to the time series. Then, we measured the model accuracy for each slice. Figure 4.6 shows that the distribution of the predicted values of the free empty parking lots per month for the year 2015 is close but not identical to the actual one. By increasing the size of the training dataset to cover years 2014 and 2015, as shown in Figure 4.8, the difference between the distributions of the predicted and actual results decrease. Now, by training the model with a larger dataset which covers three years 2014, 2015 and 2016, as shown in Figure 4.10, distribution of the prediction results for the year 2017 become almost identical to the distribution of the actual values for both ML algorithms (i.e., boosted decision tree regression, decision forest regression).

11. Data testing debt
Data for ML software is what code is for traditional software [10]. Therefore, one should track input data collection as in a rigorous manner as one should tack code changes in traditional software. In section 3.4.1, we discussed the data preparation and how the noise in data affected this stage. We also determined the selected data collection which is used while building the ML models. In this ML pipeline, the first step was preparing the ML-friendly format as input for the ML model. We tried to track the selected data input and the output for each component according as suggested by Sculley et al. [10]. It was a good practice to do some testing for each input data and how this data had affected the model. While building ML models, we found noise in data. For instance, In order to detect, such irregularities, we converted empty parking count into percentages and set 100% as the maximum value. Also, we implemented an initial data statistics to investigate the data behavior before starting actual data preparation and building ML models.

4.2 Projections for the Final Software System

As mentioned previously, ML prototypes, which we built during our case study for the prediction of empty parking lots, were initially planned to be integrated into a real-time software system, owned by Västtrafik. In this section, we will discuss the potential effects of the HTD patterns that we detected in these ML prototypes on the quality of the resulting software system and the end user experience in terms of availability, reliability, and performance. During our case study, there were HTD patterns that we could not detect in the ML prototypes. However, some of those HTD patterns are likely to be observed after the resulting software system is deployed. We will also discuss such HTD patterns in this section.

4.2.1 Impact of Observed HTD Patterns on the Software System

As mentioned in Section 3.4.1 and shown in Table 3.2, potential features that could be used to build ML prototypes consisted of the following: weather situation, day type, the percentage of the free spaces of the previous day, the percentage of the free spaces of the previous week, the percentage of the free spaces of the previous 2 weeks, the percentage of the free spaces of the previous 12 Hours, status of the day after and before the holiday day, the day name and time for each specific event. During our case study, we built ML prototypes by incrementally adding each of these features. In this section, we will consider the ML prototype that was built using all eleven features and assume that this prototype will advance to productization stage to be integrated into the real-time software. This scenario is also depicted in Figure 4.12.

During the development of ML prototypes, we used data which was collected previously. However, we will need to continuously collect and store data in order to improve further releases of the ML-based software system. This system will collect required data from different APIs such as API of the weather forecast application to collect the weather situation and Västtrafik API to collect the historical data for the percentage of the free spaces for each event. All these data will need to be stored. Moreover, during inference (i.e., while trained model makes a prediction) besides model training, the system will require live streaming of data such as weather situation, and time of the day, while remaining data might be retrieved once and stored in the database. Live streaming of these features will require a stable connection, since unavailability of one feature due to a problem in live data streaming would deteriorate system reliability and performance. On the other hand, our case study results show that each of the features related to weather situation and exact time of the day are ϵ-features and they are also among bundled features. For instance, the weather situation does not have a significant effect as shown in Tables 4.2 and 4.3. ϵ-features and bundled features are among the HTD patterns that we observed in ML prototypes we built during our case study. Moreover, using different API packages to collect such data from different resources and that will cause an increase in the glue code and pipeline jungles, which were also among the HTD patterns we detected in our case study. In addition, removal of all bundled features reduces costs imposed due to data storage demands. This is one example for explaining why detection and removal of HTD patterns in the early prototyping stage is important.

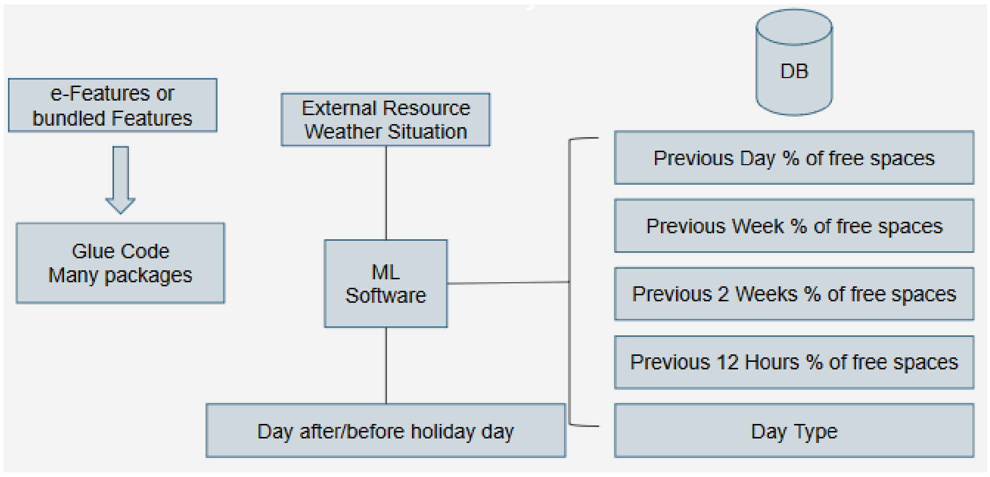

e-Features or bundled Features
Glue Code Many packages
External Resource Weather Situation
ML Software
Day after/before holiday day
DB
Previous Day % of free spaces
Previous Week % of free spaces
Previous 2 Weeks % of free spaces
Previous 12 Hours % of free spaces
Day Type

Figure 4.12: Expected ML prototype on the final software system with the potential features.

4.2.2 HTD Patterns to be Observed on the Deployed Software System

1. Correction cascades
Current ML models predict empty parking lots in Gothenburg. However, making further calibrations or using filters Västtrafik might start to use these models for other cities in the county of Västra Götaland. In this case, any change in the original ML models will affect models calibrated for other cities.

2. Undeclared consumers
If the output produced by the resulting software system, which contains our ML model, is consumed by other systems and then our system quality will influence all those systems. This is an instance of “undeclared consumers”, which is an HTD pattern that we could not observe in our ML prototypes. However, it is very likely to observe this HTD pattern once our ML model is integrated into the real-time application since information about empty parking lots in the city would be information useful for many people, hence provided by many applications as additional information that is obtained through Västtrafik API.

3. Unstable data dependencies
This HTD pattern occurs due to the fact that data/signals fed into ML models change behavior over time causing detrimental effects on the performance of ML models. Changes in the behavior of data can happen implicitly or explicitly. As previously mentioned in subsection 4.1, features C (i.e., % of free spaces previous day) and D (i.e., % of free spaces previous week) have significant effects on the prediction performance of ML models, while remaining features (except for feature B) are bundled and ϵ-features. Hence, features C and D will be used to train ML models that will go under production and that will be integrated into the final software system. Values for the number of free spaces for the previous day and previous week are collected through sensors and stored in database systems owned by Västtrafik.

"Unstable data dependencies" HTD pattern might occur implicitly, if the behavior of data changes over time. On the other hand, if there is anything wrong in sensor readings, once such problems are detected and sensor calibrations are done, this will result in the explicit occurrence of unstable data dependencies HTD pattern. Explicit occurrence of this HTD pattern occurs due to engineering ownership of the input data being separate from the engineering ownership of ML models. Sensor calibrations will have detrimental effects on the performance of ML models, since these models would have been previously trained using mis-calibrated data as input. One possible solution to tackle this HTD pattern might be giving version numbers to data (signals) before and after such significant changes in data behavior.

Features weather temperature and weather situation (i.e., feature set A) are altogether an example of bundled features, and each of them is an example of ϵ-features. If the HTD patterns bundled features and ϵ-features are not completely eliminated from the ML model that will go under productization, these patterns will propagate to the final software system. As mentioned in section 4.2.1, the software system will need to collect weather data from API of a weather forecast application. In addition to the previously mentioned reliability and performance issues, depending on the prediction models employed by the weather forecast application, behavior of the weather forecast data might change. As a result, in this situation, propagation of underutilized data dependencies (e.g., bundled features, ϵ-feature) will lead to the emergence of the HTD pattern unstable data dependencies.

4. Legacy features
In the coming releases of ML models, more features might be added to training/test dataset. As a result, features that currently have a significant impact on the prediction performance of the models (e.g., day type, % of free spaces previous day and previous week), might become redundant. It is crucial to eliminate redundant features since they increase configuration debt and data testing debt as well as leading to HTD patterns, unstable data dependencies, and upstream producers. However, eliminating legacy features from the deployed system might be challenging if the system architecture is brittle. In such a case, keeping previously added features and not adding new features that make previously added ones redundant might be a solution. Depending on the extent up to which newly added features contribute to prediction performance of ML models, a tradeoff might emerge.

5. Direct feedback loops
Front-end engineers decide to show in the user interface the parking lots in ascending order based on the percentage of empty spaces. Moreover, on the first page, for instance, only the top 5 parking places might appear. Then, this might bias (i.e., force) the users to select places that are ranked top rather than the places they actually want to park their cars. This in turn might affect the data that will be used for training of future releases of the ML model. However, this can be prevented by SE approach such as filtering results according to other criteria which makes parking lots desirable for the user. Among criteria that have the potential to play

a significant role in users' preferences might be whether the parking lot is open or closed, the distance of the parking space to users' current location, traffic situation on the way from user's current location to the parking lot. In order to identify such criteria requirements elicitation might be useful.

6. Hidden feedback loops
HTD pattern "hidden feedback loops" might be observed in the resulting software system. One possible scenario would be that if users of the parking lots prediction software also regularly check the traffic App, which makes false predictions with high false positive rates, this might prevent the user prefers the parking lots that are in locations where traffic App shows as locations with heavy traffic. As a result, users' behavior would be biased by the traffic App, which in turn changes the future training data of our parking lots prediction system, hence the behavior of this ML system as well.

7. Prototype smell
As we did during this case study, it is convenient to test new ideas in small scale prototypes. However, relying solely on a prototyping environment might result in a brittle and difficult to change software system. It is crucial that prototype models are not used directly as production code, and the prototype environment should be kept separate from the production environment. Therefore, if ML models' prototypes are not re-implemented as production code with improved abstractions and interfaces, it will be difficult to update ML module of the software system for further releases. Since this is a challenging task, there is the risk of observing prototype smells during the lifecycle of the software system.

8. Thresholds in dynamic systems
Since users will have access to the software system through Västraffik App, front-end engineers will set a threshold in order to limit the number of available parking lots in Gothenburg that will be shown in the user interface. This may result in showing no available parking places at all or list being too long to be presented on the screen. One simple way might be to put a limit in the number of available parking places to be shown and sort the available parking lots in the list in ascending order according to the percentage of free spaces.

9. Upstream producers
Data (e.g., % of free spaces previous day and previous week) will be fed to ML models from up-stream producers for training purposes of the future releases of the software system. Hence, if there is an error regarding the collection and/or generation of the data provided by any of the upstream producers, this will affect the trained ML models. Moreover, this error will propagate down-stream, if the output of our ML models is used by other applications. This is quite likely, since information about empty parking lots in the city would be information useful for many people, hence it will be provided by many applications as additional information that is obtained through Västtrafik API. In addition, if bundled/ϵ-features weather temperature and weather situation are propagated to production code for the ML models and hence,

to the final software system, a weather forecast application will serve for the system as an upstream producer. As a result, in this situation, propagation of underutilized data dependencies (e.g., bundled features, ϵ-feature) will lead to the emergence of the HTD pattern of upstream producers.

10. Reproducibility debt
In the deployment phase and after the integration stage was done, we cannot depend on the initial conditions which existed during the development of ML models. Many conditions might be changed, such as adding or removing features, changing features formats, and adjusting ML configurations. Moreover, developing the ML system with rigorous conditions is not possible, due to changes in external conditions that might introduce changes in the behavior of input data. Therefore it will be challenging and sometimes not possible to reproduce the results that are similar to prototypes' output after deployment.

11. Process management debt
Currently, software system owned by Västtrafik does not have many ML models running simultaneously. However, in the near future if dozens of ML models run simultaneously, updating configurations of many ML models manually will become a challenge. Detecting blockages in the ML pipeline will require visualization tools for monitoring the system as well as automatic recovery from blockages. Moreover, assigning resources (e.g., computing power, memory) will require making decisions based on business priorities.

12. Cultural debt
A single person, who is a software engineer with six years of software development experience and one year of experience in building ML models, worked on the ML implementations. Hence, we could not observe cultural debt thoroughly. Observation of cultural debt requires analysis of communication and interaction among members of teams consisting of data scientists, software, and systems engineers working on the development of ML-based software. We will discuss this in the next Chapter, where we present the results of the workshop that we designed and conducted.

4.3 Unobserved HTD Patterns

Plain old data type smell
We could not observe this HTD pattern, since the tool we are using to develop ML models (i.e., Azure ML Studio) has a drag and drop user interface and it does not allow the user to access the source code for ML algorithms. Therefore, we cannot state whether this HTD pattern exists in our prototypes or not.

5
Interactive Workshop Methodology and Results

5.1 Workshop Aims

In order to answer research questions RQ2 and RQ3, we conducted an interactive workshop. Since our case study was conducted for a specific case (i.e., empty parking lots prediction for Västtrafik), we aim to use the answers obtained for RQ2 and RQ3, in order to investigate the generalizability of our case study findings and complement them whenever possible. The practitioners who attended to our interactive workshop consist of data scientists and software engineers.

5.2 Workshop Design

The workshop had two parts. In the first part we explained the HTD patterns mentioned by Sculley et al [10]. In the second part we had open discussion with the practitioners to understand their opinion and perspective about the patterns. Moreover, all the practitioners belong to the same field, but they are working in different domains and the workshop idea is still an open area to discuss.

Presentation The presentation was divided into four stages; **the first stage** consisted of an introduction, the workshop objective, and detailed explanation for HTD patterns. In this part, we tried to clarify the HTD patterns as it has been mentioned by Sculley et al. [10] by using different real examples such as recommender systems for online shops and autonomous driving vehicle. Those patterns have been divided into five groups, which are complex model erode boundaries, data dependencies, changes in the external world, system-level spaghetti, and other managerial debt. We followed the division approach, as recommended by Sculley et al. [10].

The second stage was cumulative voting. The purpose of this stage is to rank the HTD patterns categories using the $100 method. Each practitioner of the voting had a virtual $100 to spend between those five HTD pattern groups according to his/her understanding from the first stage. Figure 5.1 illustrates an example of the HTD patterns questionnaire. The practitioners can spread any amount between 0 to 100 on any of the categories, but by the end of the voting, the practitioner should spread $100 in total.
We collected demographic data of practitioners (e.g., job title, company domain,

years of experience in current occupation, ML knowledge level).

The five categories of HTD patterns as Sculley et al. [10] state:

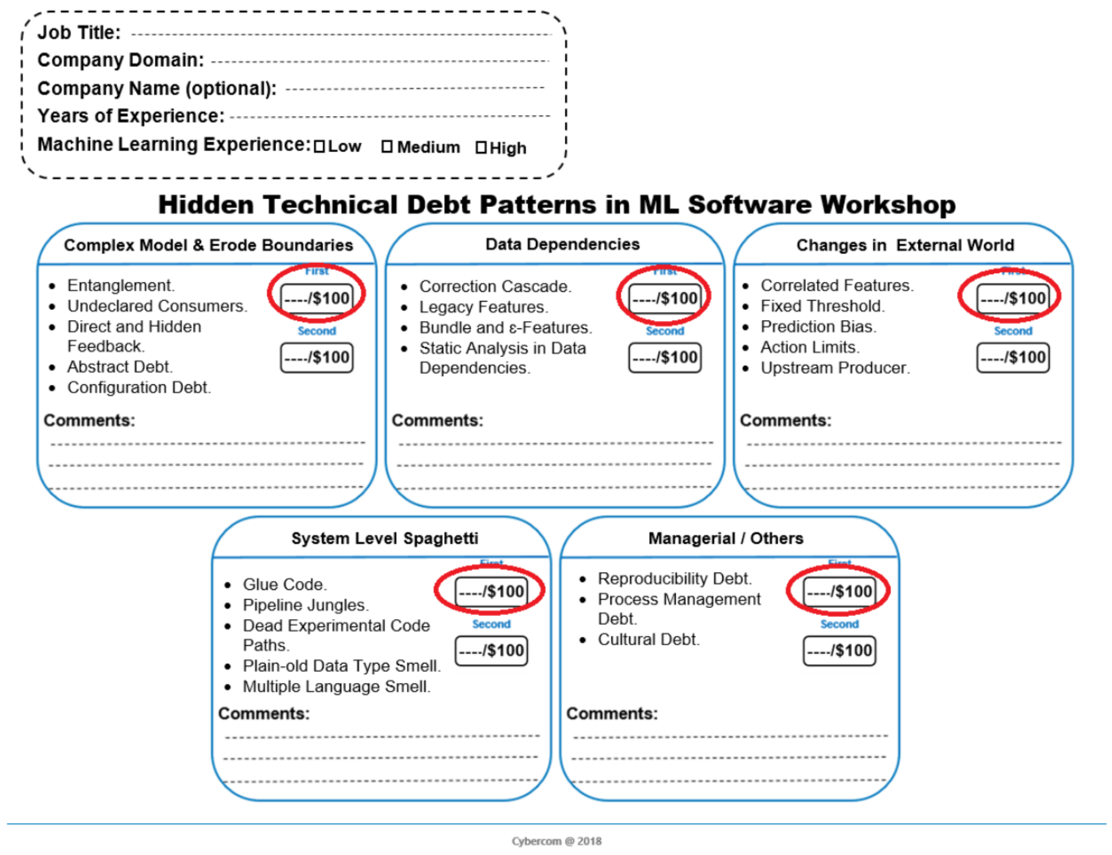
Group 1 [Complex Model Erode Boundaries]
- Entanglement
- Configuration Debt
- Undeclared Consumers
- Direct Feedback Loops
- Hidden Feedback Loops
- Abstraction Debt

Group 2 [Data Dependencies]
- Correction Cascades
- Legacy Features
- ϵ-Feature
- Bundled Features
- Static Analysis of Data Dependencies

Group 3 [Changes In External World]
- Correlated Features
- Fixed Thresholds in Dynamic System
- Prediction Bias
- Action Limits
- Upstream Producers

Group 4 [System-level Spaghetti]
- Glue Code
- Pipeline Jungles
- Dead Experimental Codepathes
- Common Smells
- Plain-old Data Type Smell
- Multiple Language Smell
- Prototype Smell
- Data Testing Debt

Group 5 [Other / Managerial Debt]
- Reproducibility debt
- Process Management Debt
- Cultural Debt


Job Title: ---------------
Company Domain: ---------------
Company Name (optional): ---------------
Years of Experience: ---------------
Machine Learning Experience: ☐ Low ☐ Medium ☐ High

Hidden Technical Debt Patterns in ML Software Workshop

Complex Model & Erode Boundaries
- Entanglement.
- Undeclared Consumers.
- Direct and Hidden Feedback.
- Abstract Debt.
- Configuration Debt.

First ----/$100
Second ----/$100

Comments: ---------------

Data Dependencies
- Correction Cascade.
- Legacy Features.
- Bundle and ε-Features.
- Static Analysis in Data Dependencies.

First ----/$100
Second ----/$100

Comments: ---------------

Changes in External World
- Correlated Features.
- Fixed Threshold.
- Prediction Bias.
- Action Limits.
- Upstream Producer.

First ----/$100
Second ----/$100

Comments: ---------------

System Level Spaghetti
- Glue Code.
- Pipeline Jungles.
- Dead Experimental Code Paths.
- Plain-old Data Type Smell.
- Multiple Language Smell.

First ----/$100
Second ----/$100

Comments: ---------------

Managerial / Others
- Reproducibility Debt.
- Process Management Debt.
- Cultural Debt.

First ----/$100
Second ----/$100

Comments: ---------------

Cybercom © 2018

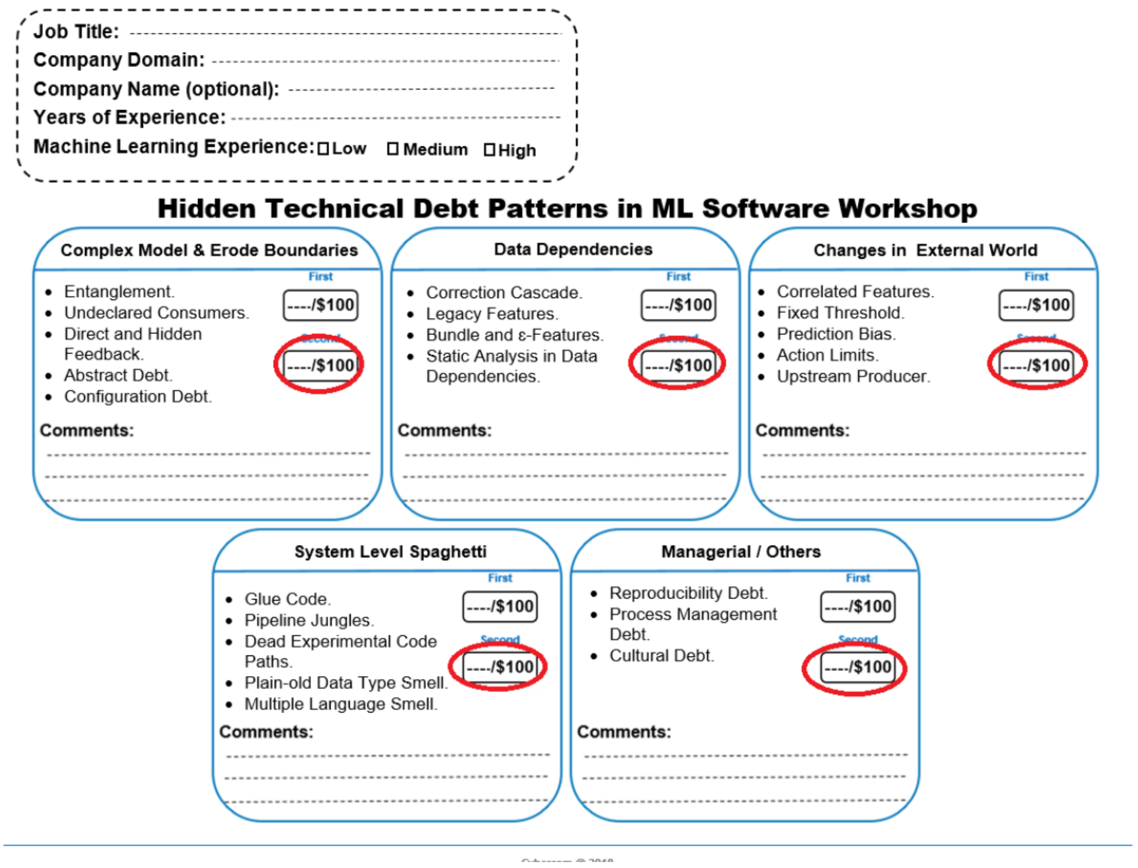
Figure 5.1: HTD pattern $100 questionnaire (round 1).

The third stage was to explain how the HTD patterns were observed in the parking lots prediction system we created for Västtrafik. We have mentioned the way to detect those patterns where we followed the detection approach as recommended by Sculley et al. [10]. Besides, we tried to explain the reasons for being unable to detect one of HTD patterns and also clarify the effects of HTD patterns after Integration of ML model into a real-time system.

The fourth stage was a second round on voting. With this second round of voting, we wanted to test if the practitioners changed their opinions about the importance of the patterns after being presented with the Västtrafik case study. In this case, the practitioner should use the second box to spread $100 value as it has been shown in Figure 5.2.

Open discussion area was the reflection stage where it was an open discussion to collect more information from the practitioners regarding their experience and the information obtained from the discussion allows us to answer RQ2 and RQ3.

The workshop materials are an introduction video to enter the practitioners in the context and an interactive presentation. The used examples are commercial and practical examples such as online bookshop and autonomous vehicles.


Job Title: --

Company Domain: --

Company Name (optional): --

Years of Experience: --

Machine Learning Experience: ☐ Low ☐ Medium ☐ High

Hidden Technical Debt Patterns in ML Software Workshop

Complex Model & Erode Boundaries

- Entanglement.
- Undeclared Consumers.
- Direct and Hidden Feedback.
- Abstract Debt.
- Configuration Debt.

First ----/$100

Second ----/$100

Comments:

Data Dependencies

- Correction Cascade.
- Legacy Features.
- Bundle and ε-Features.
- Static Analysis in Data Dependencies.

First ----/$100

Second ----/$100

Comments:

Changes in External World

- Correlated Features.
- Fixed Threshold.
- Prediction Bias.
- Action Limits.
- Upstream Producer.

First ----/$100

Second ----/$100

Comments:

System Level Spaghetti

- Glue Code.
- Pipeline Jungles.
- Dead Experimental Code Paths.
- Plain-old Data Type Smell.
- Multiple Language Smell.

First ----/$100

Second ----/$100

Comments:

Managerial / Others

- Reproducibility Debt.
- Process Management Debt.
- Cultural Debt.

First ----/$100

Second ----/$100

Comments:

Figure 5.2: HTD pattern $100 questionnaire (round 2).

5.3 Workshop Result

There are different opinions about the required number of practitioners in the workshop since this workshop requires more interactive discussions, so the preferable number of practitioners is between 5-10 people from different domains. Actually, this workshop has practitioners who belong to different fields with a variety of experience level in both sides; the industry and the machine learning. Table 5.1 illustrates an overview of the practitioners in the workshop.

Table 5.1: Workshop practitioners details.

Job Title	Company Domain	Years Experience	ML Experience
Junior Data Scientist	Telecom and Software	2	Medium
Application Developer	Software Solution	3	Low
CTO	Software Solution	> 10	Low
Data Scientist	Telecom Solution	8	High
Data Scientist	Legal Services	>15	High
Senior QA	Product Customer Management	9	Low

There are no significant changes in the obtained results for the first and the second rounds of the $100 questionnaires in the workshop. Thus, Tables 5.2 and 5.3 illustrate an overview of the $100 questionnaire results, which represent the ranking for five groups of HTD patterns, during the first and the second rounds in the workshop.

Table 5.2: Workshop practitioners results for $100 questionnaire (First Round).

Group 1	Group 2	Group 3	Group 4	Group 5	Total
Complex Model	Data Dependencies	Changes In External World	System-level Spaghetti	Other Debt	
17.5%	20%	26.6%	23.4%	12.5%	100%

Table 5.3: Workshop practitioners results for $100 questionnaire (Second Round).

Group 1	Group 2	Group 3	Group 4	Group 5	Total
Complex Model	Data Dependencies	Changes In External World	System-level Spaghetti	Other Debt	
15.8%	21.7%	26.7%	23.3%	12.5%	100%

The practitioners have focused on the "Changes In External World" and "System-level Spaghetti" groups regarding the previous results. This is followed by the category “Data Dependencies”. In the next subsection, we will summarize the results we obtained through analysis of the qualitative data we gathered during the workshop.

This topic is too new to the practitioners, some of them probably did not understand the concept completely and hence they might not have been sure about the answers they gave us during the workshop. Moreover, to assessing the importance of HTD patterns requires taking into account the whole software development life cycle (SDLC) rather than only focusing on prototyping or production. However, due to their specific roles (e.g., Data Scientists, Software Engineer), most practitioners could only focus on a specific phase of ML software development. This prevented them from understanding most HTD patterns and the implication of their emergence in the products.

5.3.1 Practitioners’ Prioritization of HTD Patterns and the Rationale Behind

This section addresses RQ 2.1 and RQ 2.2 through discussing the obtained results from the interactive workshop.

Focus on Standalone ML Applications
From the perspective of the ML expert with more than 15 years of experience, HTD patterns such as undeclared consumers, unstable data dependencies, direct and hidden feedback loops did not seem important. While he was trying to explain the rationale behind his opinion, we realized that he only focuses on the ML application as an isolated standalone system. For instance, he did not consider the ecosystem

of ML systems.

> *"Yeah, maybe. Maybe, I am just like failing to see the overall. Because we have similar things like that. We do like, let's say we have machine learning model for one customer, for another customer, for the other one and they are trying to see to learn from those models. But for us, this is a machine learning problem. This has nothing to do with software. Well, it has to do something with software engineering, of course, but it is a pure machine learning problem."*

Another interesting observation was that he perceived us as if we were talking about "ensemble methods", which is a machine learning technique that combines several base models in order to produce one optimal predictive model. However, we were talking about an ecosystem, where multiple ML systems run simultaneously depending on each other or independently.

> *"I know but this is not software versus machine learning. You just basically talking about an ensemble method you know about how you choose or how you combine them. This is not software versus machine learning. This is 2 machine learning problems."*

The direction of the discussion started to change after asking him what will happen if there are multiple machine learning models with the high dependency between them and all these models integrated into one software system.
Obviously, the practitioners have focused on the "Changes In External World" and "System-level Spaghetti" debt groups. Part of them agreed with that changing in the features weights over years is possible and it is not an easy task to catch those changes. In another opinion, depending on the obtained features from the historical training data set is enough and the changes in the features weights will be slight.

Analyzing the behaviour of ML Proof Of Concept
Part of the practitioners does not have agreed for studying these patterns since they supposed that all these studies should be implemented during preparing the ML model. But, all of them have focused on small machine learning project which is a standalone ML model.
One of ML experts with more than 15 years of experience in ML, strongly indicated that HTD patterns in the category "Data Dependencies" such as legacy, $\epsilon-$ and bundled features are not an important issue and this has nothing to do with software engineering. However, the main reason about his tendency to see these issues as pure Machine Learning problems is due to his perception. During the workshop, when we explained the importance of feature selection and avoiding legacy, $\epsilon-$ and bundled features, he misunderstood that we recommend ML experts be given in advance the right features to build ML models.

> *"Regarding which features he uses, it's his work. He decides which features to use. He won't be given the features. You don't tell him*

> *you just only use those three features. You are giving him: 'here is the data, here is your problem. Find the solution.' He comes with the features, ...*"

The main reason of his misunderstanding was that he mostly focused on the early prototyping phase of the ML software rather than the whole Software Development Life Cycle (SDLC). For instance, he did not consider the development of production code by software engineers based on the prototype that ML expert built in the prototyping phase. As a result, he could not see that glue code and configuration debt can be introduced in the production code due to extensive number of features used. We reminded him that SDLC consists of prototyping, production code development, deployment, re-training (on-site) followed by development of versions with improved ML models. We also indicated that after deployment due to some features, the resulting ML software might be dependent on other external systems providing some features as input to the ML software. He occasionally agreed with us when we reminded these facts to him. But, during most of the discussion session, he showed a tendency to see all these HTD patterns only through the lens of Machine Learning, but not also through the lens of Software Engineering. He also agreed about the existence of a tradeoff between number of features used to increase prediction performance and allocated resources that an ML expert usually considers (e.g., amount of memory used, training time, testing time).

> *"This is a question that we every time, all the time we ask ourselves. Is it worth it, like should we add something that will increase my... I don't know, accuracy with 2% to 1%, but if we add all those, like a lot of memory, I need longer training time, longer prediction time. These are the natural trade off that you need to take on the way."*

However, regarding the HTD category of "Data Dependencies", another ML expert, who worked in several companies from very different domains (e.g., ad-click advertisement systems, telecommunication, etc.) emphasized the importance of data dependencies.

> *"I mean for example, take the example of data dependency. We have experience like things are like product name has changed, even like the column has been renamed differently, then you need to like sort of, first of all you need to know about it. If someone changes without telling you, then there is no way you are going to know. But then of course, you need have somehow an alert system, if there is a discrepancy then you should get notified somehow."*

HTD Patterns Regardless of Domain

The practitioners agreed with that most of the technical debt patterns groups are observed during their job. Part of them does not have any problem to handle this technical issue after the deployment process. They do not mind also that dealing with ML models as a black-box without any analysis for the correlation between ML

models and the container software system. In contrast, part of them is looking for a possible solution to eliminate these technical pitfalls and control the production and the maintainability process especially with large complex systems which run multi-machine learning models at the same time.
Another interesting finding is that most practitioners agreed that HTD framework proposed by Sculley et al [10]. is applicable to other domains besides ad-click advertisement systems Sculley and his colleagues at Google Inc. are specialized at. This was especially emphasized by the ML expert with 8 years of experience in total in various industries including telecommunication systems and ad-click online advertisement systems.

> *"I worked with a lot of industries doing machine learning... I can list at least 3 different industries, like online marketing,... I think this [implying HTD] is really useful. That's why I am here [implying the workshop]... "*

5.3.2 Practitioners' Recommendations to Manage HTD Patterns

This section will try answering RQ 3 through discussing the obtained results and the suggested practitioners' opinions in the interactive workshop.

- **Are the culture and managerial debt a real obstacle to develop and deploy high-quality ML software?**
  The used example to describe the last group debt was the autonomous driving where there are multiple engineers to produce this product such as embedded system engineers, software engineers, machine learning experts, computer vision experts, integration engineering and so on. Each domain has specific properties like tools, programming language, practices, and frameworks. And the question was how those engineers will communicate to develop, deploy and maintain this type of product?
  Most of the practitioners' opinions were that there is no problem for the communication between the engineers' teams if the project and teams size are quite small. They have also mentioned that each team will deal with the outputs of other teams as black-boxes. Another question in this period was fired if the project is a large scale ML software and we used the same example autonomous driving. This system contains multiple ML models for different purposes like data analytic for saving useful sensors data in the cloud, another model for image processing and many systems will communicate with each other. Also, this type of project requires large teams from different technical background to support and deliver the project. One of the practitioners suggested looking for a standard generalized language to unified and simplify the communication among the different teams. However, all the practitioners agreed with that detection with a rational expectation of the technical debt during the initial developing stage is an effective way to enhance the deployment procedure and decrease the maintainability cost in the long term.

ML expert who currently works in telecommunication industry indicated that communication between the software engineers and ML experts is crucial to build right ML models.

> *"... You need to have a sort of communication with the development team. That's covered in your changes in the machine learning model, just because of that. That's a simple example. With other story, like if you change two different kind of products, then by nature they'll have two different properties. And when you are trying to predict something, then of course the features that are used needs to be changed accordingly and things like that."*

She also indicated that her current team consists of 5-6 ML experts and the whole team of ML experts in the entire company consists of 20 or 30 people. Due to the set up in the company, she indicated that she does not encounter huge problems. However, regarding her experience in start-ups she has different views:

> *"I worked in a startup company. So, in start-up companies, you are trying to collaborate too much more with software engineers. And if they are not coming from data, then it will be slightly hard for them to comprehend what you need. That's common I would say."*

However, regarding communication between software developers and ML experts in startups, ML experts who is a junior data scientist with 2 years of experience, had a different view. He currently works in a startup as a member of a team consisting of 3 software engineers and 3 data scientists. He indicated that they established an efficient and effective way of communication.

> *"... it is a very small company. It's 11 people, but it's very healthy how they are like you bring up a new model like the one we found the other day, there is random forest regression model that has actually linear regression as leaf node. So, in the end, it can actually, predict value it hasn't seen before, which is the problem with random forest, they can't do that, normally. So, we want to implement this. So, ... we kind of made a presentation to the software developers, and we are discussing like how this works and how they are going to implement it. It is their case, right, because they have already built the whole data software pipelines themselves, the code base."*

"It is the problem of communicating. It's like, even in larger scale, you have to bring people together that should be together. . . . "

- **The suggested practices, tools and frameworks to detect and eliminate HTD**
  ML expert who is a junior data scientist with 2 years of experience mentioned that using Lean Software Development (LSD) is a good practice to build ML software with high quality. The initial step is creating the prototype of ML software architecture by ML experts to optimize the ML model, then the implementation stage will start by ML and software developers where they put in their consideration the quality from two perspectives which are the ML and software attributes.
  Another participant has suggested that establishment of alerts systems for traceability of data dependencies within an organization, since sometime some ML developers change the features names or scale and over time the data loses the consistency. Also, start depending on the web services APIs as a standalone resource of data for more abstraction and less data dependency where ML model can consume the required input data via these APIs.

6
Threats to Validity

In this section, we will discuss the threats to construct, internal and external validity. In the following subsections, we explain each of these threats to validity for both the case study and the workshop that were conducted in order to answer the research questions, which this thesis aims to answer.

6.1 Construct Validity

Construct validity refers to how well operational measures represent what researchers intended them to represent in the study in question. In order to circumvent threats to construct validity, as a construct for maintainability, we used the HTD framework proposed by Sculley et al. [10].

6.1.1 Case Study

Some HTD patterns were not detected by the investigator during the case study. However, we thoroughly followed the detection approach as recommended by Sculley et al. [10]. Moreover, we discussed further HTD patterns that are to be observed after the deployment of the ML intensive software product in 4.2.

6.1.2 Workshop

The perspective of the practitioners regarding the HTD patterns, including the prioritization and motivation, might not serve or fulfill what this study is investigating or requiring. To mitigate this, we conducted a workshop to explain the HTD patterns to the practitioners. Also, we discussed the data which we obtained from the case study and called for an open discussion to collect practitioners' perspective.

6.2 Internal Validity

In this subsection, we explain internal threats to validity regarding the case study and the workshop as well as how we circumvented some of these threats.

6.2.1 Case Study

The subject and the observer are the same person who worked on the ML software development and investigated the HTD patterns.

While building the ML models, which were later analyzed for HTD patterns, the subject did not have any knowledge about the Hidden Technical Debt framework by Sculley et al. [10]. Therefore, maturation of the subject was only after building those ML models and during the period when the subject conducted further experiments consisting of applying data analysis and ML techniques in order to explore HTD patterns in the previously built ML models. Moreover, there were no events that took place leading to any introduction of HTD patterns consciously or unconsciously.

6.2.2 Workshop

In the workshop, the practitioners tried to prioritize HTD patterns by using $100 method. But, the obtained results are very close and we could not get a very clear conclusion. To get more accurate results we called an open discussion to collect practitioners' opinions.

The practitioners have different ML experiences and different technical backgrounds. This might have affected their evaluation. However, to mitigate this we introduced the topic to all the practitioners in order to let them have the same knowledge about HTD patterns. In order to ensure that every attendant understands the concept of HTD, we used examples of ML applications that are encountered in our daily lives. Moreover, in order not to bias the attendants with our findings, which consisted of the HTD patterns detected in the ML models built for empty parking lots prediction, we applied $100 method twice. $100 method was applied first, after explaining the general concept of HTD and was applied the second time after explaining our case study findings. Practitioners' responses did not change when $100 method was employed the second time.

6.3 External Validity

In this subsection, we explain external threats to validity regarding the case study and the workshop, as well as how we circumvented some of these threats.

6.3.1 Case Study

Different assessing tools might have led to different results, the used tool is only one tool, which is Microsoft Azure Machine Learning Studio. ML cloud solutions are drag and drop tools beside to use a ready pipeline with the custom ML algorithms. It is not easy to debug and track the errors in those systems, then probably that causes hidden technical issues, and it will not be easy to catch.

By design, case studies have a very limited external validity stemming from the fact that a topic is studied within its context. The context of this thesis is in a specific domain, which is empty parking lots prediction that aims to predict a total count of empty parking lots for a given date and time slot. Different results might be obtained if a different context is studied. Moreover, the subject is a software engineer with eight years of software development experience, and recently he has started building ML prototypes.

Different practitioners with different expertise and backgrounds could evaluate the patterns differently. Therefore, further replications of this case study might be required.

6.3.2 Workshop

The attendants of the workshop consisted of 6 practitioners, each being a member of a specific category of practitioners. These categories were the following:

- Data scientist having high experience in ML and specialized only in a specific ML application domain.
- Data scientist having high experience in ML and has worked in various ML application domains.
- Data scientist with a software/systems engineering background and low experience in ML.
- Software developer specialized in mobile applications.
- Quality assurance manager, who should care for the quality of the software product.
- CTO who is planning for ML algorithms to become an integral part of their software products.

Hence, regarding the attendants of the workshop, our sampling of subjects was not random, but the external validity of our results is extended through the *impressionistic instance model*. Therefore, our results can only be generalized only for those categories.

6.4 Dependability

As Feldt and Magazinius define in [27], dependability concerns the extent to which the operations of a study can be repeated by other researchers, achieving the same results. We detailed the research methodologies employed in this thesis and thoroughly described the data collection and analysis procedures. By doing so, we think this study can be reproduced by other researchers.

7
Conclusion and Future Plan

This section contains two sections; the first one will discuss the conclusion, which includes the contribution of this thesis. The second section will discuss the future plan for this research.

7.1 Conclusion

This thesis aims to explore the emergence of Hidden Technical Debt (HTD) patterns in ML systems during the early phase of Software Development Life Cycle (SDLC), namely "prototyping phase". HTD patterns framework was proposed by Sculley et al. [10] to address maintainability issues in ML software, based on their experience gained through developing online ad-click systems at Google over a couple of decades.

In order to fulfill the goals of this thesis, we conducted a case study followed by an interactive workshop. During our case study, we examined HTD patterns in ML models that were developed to predict empty parking lots for Västtrafik. Hence, our case study also aimed to explore up to what extent the HTD framework by Sculley et al. [10] is applicable to an ML system other than online advertisement systems. In order to assess the generalizability of our case study findings, we later conducted a workshop with practitioners consisting of data scientists and software engineers.

Regarding the results of our case study, out of 25 HTD patterns, we could not observe only 1 HTD pattern, which is "plain old data type smell". The reason is that the framework we used to develop ML prototypes (i.e., Azure ML Studio) has a drag and drop interface and does not give the user access to the source code, which is encapsulated in ML modules. However, the fact that we could not observe this HTD pattern does not imply that it does not exist in the ML models we developed for our case study. On the other hand, we were able to detect the following 12 HTD patterns in the ML models: entanglement, bundled features, ϵ-features, correlated features, glue code, pipeline jungles, dead experimental code paths, abstraction debt, multiple language smell, configuration debt, prediction bias, and data testing debt. Our analysis results show that if these HTD patterns are not tackled during the prototyping phase, they will propagate to later stages of ML SDLC and also to further releases of ML software. Moreover, they will continue to build up further and become more difficult to fix. Finally, if they are not detected and fixed during the prototyping phase, some of these prototyping related HTD patterns can also trigger the remaining HTD patterns, which have the potential to emerge during productization or deployment. **To summarize, detection and removal/management of these HTD patterns as early as possible in SDLC (i.e., prototyping stage) is crucial**. During our case study analyses, we also projected our findings to production and deployment phases when the final software system that will be obtained once ML models are integrated into Västtrafik's real-time software system. Our projection analyses results indicate that all remaining 12 HTD patterns could only be observed in the deployed ML software system.

Our workshop results show that only high level of expertise in ML may not be enough to grasp how detection and removal/management of HTD patterns can be related to software engineering to improve the maintainability of ML software, if practitioner focuses only on prototyping phase ignoring overall development lifecycle. Moreover, our workshop results also support the generalizability of our results to various ML algorithms and application domains to some extent. Data scientist who worked in different ML applications (e.g., telecommunication, online ad-click systems, etc.) also indicated that these HTD patterns are applicable to different types of ML applications.

Amount of data and number of features that we used to train ML models, which were analyzed during the case study, are small/medium scale, and algorithms and resulting models are less complicated compared to deep learning applications. In spite of these, we were able to detect all prototyping-related HTD patterns except for one (i.e., plain old data type smell) during our case study. Moreover, as data size increases, data testing debt is more likely to emerge. Increase in the amount of data and number of features also increase the risk of emergence of the HTD patterns bundled, ϵ-, legacy, and correlated features. This -in turn- increases glue code and pipeline jungles. In addition, an increase in model complexity leads to more entanglement. As a result of these, configuration debt also increases. To summarize, our case study results indicate that **HTD patterns, which have the potential to emerge during the prototyping phase, can emerge even in less complex ML models that are built with small data size and number of features.**

Based on recommendations of D. Sculley and his colleagues at Google Inc. [10, 12], expertise we gained while conducting the case study and practitioners' recommendations we obtained during the workshop, below we provide some guidelines to practitioners that could assist them in detection and removal/management of HTD patterns in ML prototypes.

- **Test data quality:** Eliminate noise through manual checking and statistical analysis before training models. When such ML models are deployed as part of the software system, and if other applications act as "undeclared consumers" of the output produced by the ML application, those applications will be affected negatively. In such a case, the ML system will act as "upstream producer" causing the wrong prediction results to propagate down to other applications.

- **Use simple ML models with acceptable prediction performance:** Avoid using complex ML algorithms (e.g., deep learning) as long as you can achieve acceptable prediction performance results by using simpler ML algorithms. To build industrial applications, experts such as Andrew Ng and Ian Goodfellow recommend starting with the simplest viable model that gives acceptable performance results [29]. In order to decide what acceptable performance results would be, the first step is to choose target metrics (e.g., accuracy, precision, recall, error rate) and corresponding metrics values. In order to decide on the metric type and corresponding metric value that needs to be satisfied, it is crucial to discuss requirements of the ML application with business analysts and requirements engineers. For instance, in a voice recognition application high accuracy would be required, but while predicting a rare disease, high accuracy would not mean much, hence one would aim for high recall. Empty parking lots prediction models we developed for our case study would not require high accuracy, and also the problem we aim to solve is a regression problem rather than a classification problem as indicated by business analysts at Västtrafik. Choosing the simplest viable solution, helps reducing model hyperparameters so that entanglement can be managed making debugging of the resulting code easier. Moreover, simpler models help to reduce configuration debt.

- **Detect and remove/manage bundled, ϵ-, legacy and correlated features:** In order to select the features that have significant effects on the model, it is required to employ feature selection algorithms and correlation analysis. The cost of dealing with the ineffective features is quite high in both sides the maintenance and the deployment. Retrieving those ineffective features during the real-time system is costly, where the model cannot run without the whole input features. Removing those ineffective features during the early stage of development will enhance the system performance and decrease the maintainability cost in the future.

- **Use APIs to minimize glue code and pipeline jungles:** Glue code and pipeline jungles usually emerge during the data preparation stage (e.g., splitting data, preparing data as input to ML algorithms, etc.). Therefore, instead of consuming data from static components, it might be a good solution to get data using APIs.

- **Avoid using multiple programming languages:** If different programming languages are used in different stages of ML pipeline (e.g., data retrieval, preprocessing, splitting and ML algorithms implementations, etc.), then execution of integration tests to ensure the proper functioning of the ML pipelines becomes challenging. Therefore, it would be a good practice to avoid using multiple programming languages as much as possible.

- **Clean dead experimental code paths before code production:** In the production stage, some experimental paths, which have never been used or tested before, could be pushed back to the main branch in the code versioning system. These dead experimental paths could lead to making the system down, and it may not be trivial to troubleshoot causes of failure. Therefore, it is crucial to avoid branching out from the main branch for experimentation purposes. Moreover, ML experimentation/prototyping environment should be kept separate from ML code production environment. Dead experimental code paths should be removed from ML algorithms, which result from the prototyping stage, before these algorithms are refactored for the code production phase.

- **Communicate with SE and systems engineers:** As we have mentioned throughout this thesis, it is crucial to consider the whole lifecycle of ML software development. Emergence of an HTD pattern during prototyping may affect reliability and/or performance, or it can further lead to the emergence of (an)other HTD pattern(s) in the deployed ML software system. Detection of some HTD patterns (e.g., legacy features) during further releases of ML software may be too late for their removal due to the existing architecture of the software system. In order to make decisions regarding removal or management of HTD patterns detected during later releases of ML software, data scientists should have discussions with software and systems engineers.

Teams of practitioners from different technical background such as data scientists (i.e., ML experts), software engineers, system engineers, integration engineers, and test engineers should collaborate in order to build a robust integrated ML system. If there is miscommunication between these teams and each team works individually that will cause a critical problem during the integration phase, consequently in the deployment stages later. One way to tackle cultural debt is that every practitioner should be made aware that their actions in one phase of SDLC might affect (an)other phase(s), hence decisions made by practitioners with other roles. One suggested way to tackle cultural debt might be Lean Software Development (LSD): the first step is building a prototype of ML system by ML experts, and the next step is the implementation stage which will be implanted by ML and software engineers who will care for the system quality as a whole. Moreover, teams can employ practices of agile methodologies, such as having retrospective meetings after each sprint and during this meeting, they can determine the dependencies between each team.

7.2 Future Plan

This type of research is quite new, and it requires more investigation. In this research, we have focused on examining this HTD patterns through developing the initial stage of a prototype of a prediction system. As a future plan, we have an aim to analyze those HTD patterns impacts for retraining stage, for developing a standalone ML system, and for implementing an integrated ML system. In addition, today the modern AI system which contains more than one ML model at the same time has been increased and the integration process became more advanced. However, these ML models are working together inside one integrated system and delivering this kind of system without studying the impacts of the dependencies among those different models and the software system product will cost more.

As for the next step in the future plan, we decided to try reducing the gap between the software and ML development from different aspects like architecture, development, deployment and maintenance stages then delivering an integrated ML system. Generally, as an initial stage for developing any startup project, we study the software quality attributes to start building the software architecture by employing many tactics that satisfy the requirements. However, as a future vision, it is precious to study the additional attributes which should be included to build robust ML software architecture.

Bibliography

[1] Gomez-Uribe, C.A & Hunt, N, 2016, ’The Netflix Recommender System: Algorithms, Business Value, and Innovation’, ACM Transactions on Management Information Systems, January 2016, Volume 6 Issue 4.

[2] Kenthapadi, K; Le, B & Venkataraman, G, 2017, ’Personalized Job Recommendation System at LinkedIn: Practical Challenges and Lessons Learned’, The Eleventh ACM Conference on Recommender Systems, August 27-31, 2017, Pages 346-347.

[3] Sculley, D; Otey, M.E; Pohl, M; Spitznagel, B; Hainsworth, J & Zhou, Y, 2011, ’Detecting adversarial advertisements in the wild’, The 17th ACM SIGKDD international conference on Knowledge discovery and data mining, August 21 - 24, 2011, Pages 274-282.

[4] McMahan, H.B; Holt, G; Sculley, D; Young, M; Ebner, D; Grady, J; Nie, L; Phillips, T; Davydov, E; Golovin, D; Chikkerur, S; Liu, D; Wattenberg, M; Hrafnkelsson, A.M; Boulos, T & Kubica, J, 2013, ’Ad Click Prediction: a View from the Trenches’, The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD).

[5] Ning, H; Rohm, W.A & Ericson, G (2017), Overview diagram of Azure Machine Learning Studio capabilities. Retrieved from https://docs.microsoft.com/enus/azure/machine-learning/studio/studio-overview-diagramdownload-the-machinelearning-studio-overview-diagram

[6] Balzer, P, 2015, Prediction of car park occupancy with open data. Retrieved from http://mechlab-engineering.de/2015/03/vorhersage-der-parkhausbelegung-mit-offenen-daten/

[7] Alpaydin, E.(2016). Machine Learning: The New AI. Retrieved from https://mitpress.mit.edu/books/machine-learning

[8] Runeson, P & Höst, M, 2009, ’Guidelines for conducting and reporting case study research in software engineering’, M. Empir Software Eng (2009) 14: 131, April 2009.

[9] Robson C (2002) Real World Research. Blackwell, (2nd edition)

[10] Sculley, D; Holt, G; Golovin, D; Davydov, E; Phillips, T; Ebner, D; Chaudhary, V; Crespo, J.F; Young, M & Dennison, D, 2015, ’Hidden Technical Debt in

Machine Learning Systems'. Retrieved from http://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.

[11] Kruchten, P; Nord, R & Ozkaya, I (2012), 'Technical Debt: From Metaphor to Theory and Practice', IEEE Software, Nov-Dec 2012, Volume: 29, Issue: 6, Page(s): 18 - 21.

[12] Breck, E; Cai, S & Nielsen, E, 2017, 'The ML test score: A rubric for ML production readiness and technical debt reduction', IEEE International Conference on Big Data, 11-14 Dec. 2017.

[13] Besker, T.(2018) An Empirical Investigation of the Harmfulness Of Architectural Technical Debt. Gothenburg: CHALMERS UNIVERSITY OF TECHNOLOGY.

[14] Shapiro, H; Caserio, C; Martens, J; Ericson, G & Rohm, W.A, 2017, Machine learning tutorial: Create your first data science experiment in Azure Machine Learning Studio. Retrieved from https://docs.microsoft.com/en-us/azure/machine-learning/studio/create-experiment

[15] Kepuska, V & Bohouta, G, 2018, 'Next-generation of virtual personal assistants (Microsoft Cortana, Apple Siri, Amazon Alexa and Google Home)', IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC), 8-10 Jan. 2018.

[16] Hazelwood, K; Bird, S; Brooks, D; Chintala, S; Diril, U; Dzhulgakov, D; Fawzy, M; Jia, Y; Kalro, A; Law, J; Lee, K; Lu, J; Noordhuis, P; Smelyanskiy, M; Xiong, L & Wang, X, 2018, 'Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective', IEEE International Symposium on High Performance Computer Architecture (HPCA), 24-28 Feb. 2018.

[17] Martinez-Plumed, F; Avin, S; Brundage, M; Dafoe, A; hEigeartaigh, S.O & Hernandez-Orallo, J, 2018, 'Accounting for the Neglected Dimensions of AI Progress', Retrieved from https://arxiv.org/abs/1806.00610?

[18] Agarwal, A; Bird, S; Cozowicz, M; Hoang, L; Langford, J; Lee, S; Li, J; Melamed, D; Oshri, G; Ribas, O; Sen, S Slivkins, A, 2017, 'Making Contextual Decisions with Low Technical Debt', 9 May 2017, Retrieved from https://arxiv.org/abs/1606.03966.

[19] Luc, P; Couprie, C, LeCun, Y & Verbeek, J, 2018, 'Predicting Future Instance Segmentations by Forecasting Convolutional Features', Retrieved from https://arxiv.org/abs/1803.11496

[20] Huval, B; Wang, T; Tandon, S; Kiske, J; Song, W; Pazhayampallil, J; Andriluka, M; Rajpurkar, P; Migimatsu, T; Cheng-Yue, R; Mujica, F; Coates, A & Y. Ng, A, 2015, An Empirical Evaluation of Deep Learning on Highway Driving, Retrieved from https://arxiv.org/abs/1504.01716

[21] Pei, K; Cao, Y; Yang, J & Jana, S, 2017, 'DeepXplore: Automated Whitebox Testing of Deep Learning Systems', The 26th Symposium on Operating Systems Principles, October 28 - 28, 2017, Pages 1-18.

[22] Zavershynskyi, M. (2017, Jul 1). Technical Debt in Machine Learning. Retrieved from https://towardsdatascience.com/technical-debt-in-machine-learning-8b0fae938657.

[23] Ericson, G; Rohm, W.A; Mabee, D & Martens, J, 2017, Feature engineering in data science. Retrieved from https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/create-features

[24] Kim, J; Feldt, R & Yoo, S, 2018, Guiding Deep Learning System Testing using Surprise Adequacy. Retrieved from https://arxiv.org/abs/1808.08444

[25] Arpteg, A; Brinne, B; Crnkovic-Friis, L & Bosch, J, 2018, Software Engineering Challenges of Deep Learning. Retrieved from https://ieeexplore.ieee.org/abstract/document/8498185

[26] Liu, H & Motoda, H. (1998). Feature Selection for Knowledge Discovery and Data Mining. New York: Springer Science+Business Media LLC.

[27] Feldt, R & Magazinius, A, January 2010, Validity Threats in Empirical Software Engineering Research - An Initial Survey. Retrieved from http://www.robertfeldt.net/publications/feldt_2010_validity_threats_in_ese_initial_survey.pdf

[28] Zheng, A. (2014). The challenges of building machine learning tools for the masses, presented at NIPS Workshop on Software Engineering for Machine Learning, Montreal, December 13, 2014.

[29] Goodfellow, I; Bengio, Y & Courville, A. (2016). Deep learning. MIT Press Ltd.

A

Data Collection Of The Selected Features

A.1 The numerical weather summary data description

ID	Summary Description
1	Breezy
2	Breezy and Foggy
3	Breezy and Mostly Cloudy
4	Breezy and Overcast
5	Breezy and Partly Cloudy
6	Clear
7	Foggy
8	Heavy Rain
9	Heavy Rain and Breezy
10	Heavy Snow
11	Light Rain
12	Light Rain and Breezy
13	Light Rain and Windy
14	Light Snow
15	Light Snow and Breezy
16	Mostly Cloudy
17	Overcast
18	Partly Cloudy
19	Possible Light Rain
20	Possible Light Snow
21	Rain
22	Rain and Breezy
23	Rain and Windy
24	Snow
25	Snow and Breezy
26	Windy
27	Windy and Mostly Cloudy
28	Windy and Overcast
29	Windy and Partly Cloudy
30	Heavy Snow and Breezy

A.2 The numerical values for the day after and before the holiday

Day [After|Before] the Holiday	Value
Yes	1
No	0

A.3 The numerical values for the day type

Day Type	Value
Working Day	0
Weekend	1
National Events	2
School Holidays	3

A.4 The technical pipeline and components of ML model development

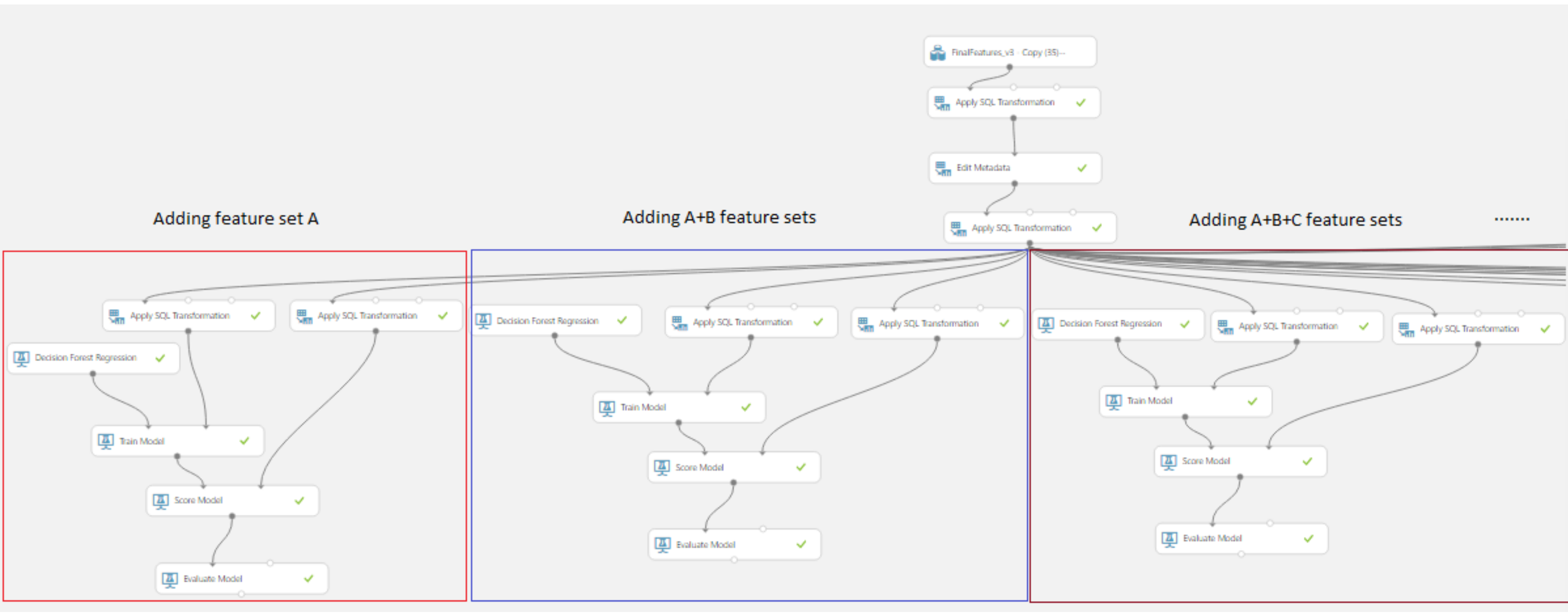
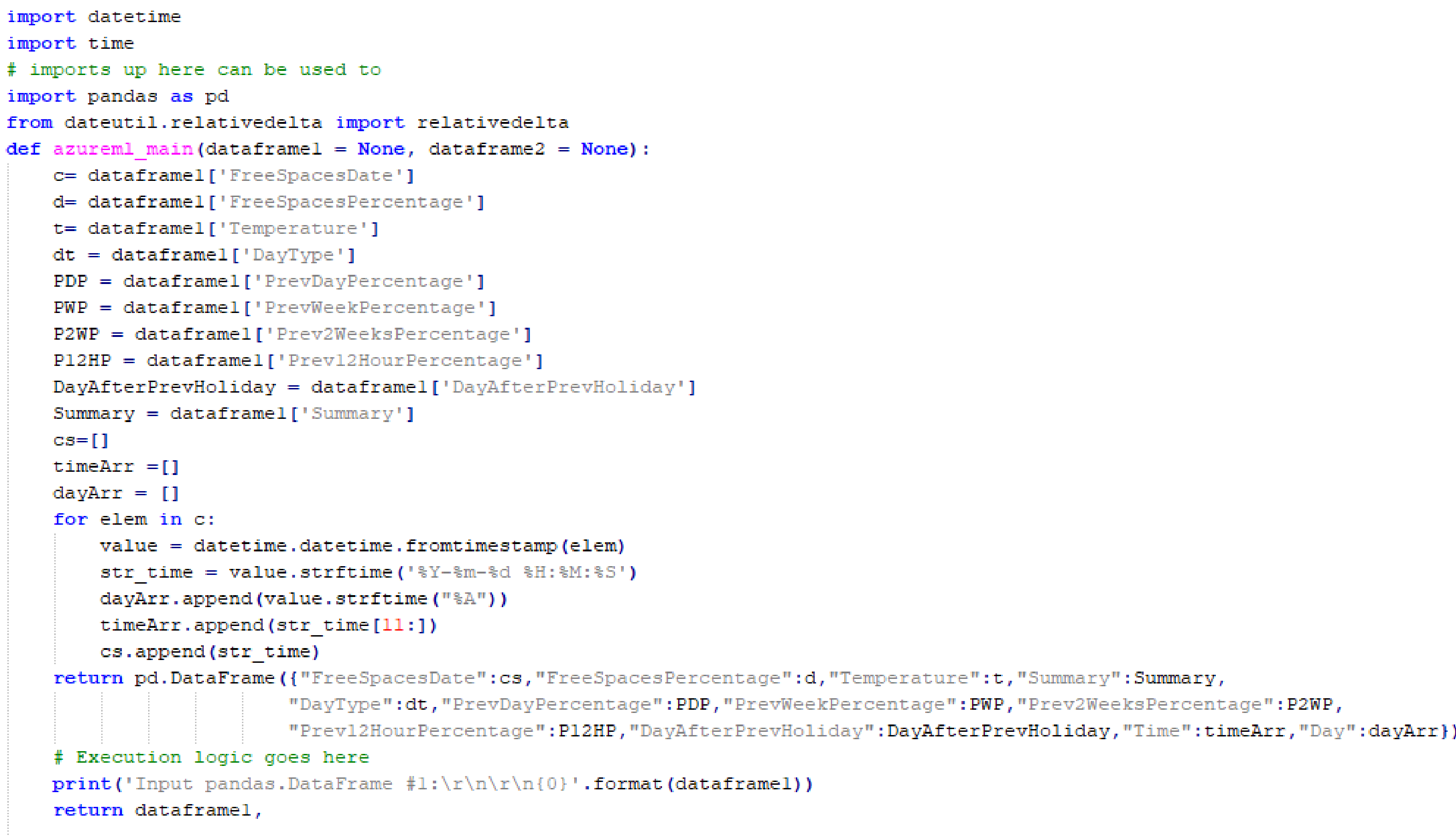


Figure A.1: Azure technical pipeline and components for ML model pipeline

A.5 Execute Python Script component used for preparing the input features.



```
import datetime
import time
# imports up here can be used to
import pandas as pd
from dateutil.relativedelta import relativedelta
def azureml_main(dataframe1 = None, dataframe2 = None):
    c= dataframe1['FreeSpacesDate']
    d= dataframe1['FreeSpacesPercentage']
    t= dataframe1['Temperature']
    dt = dataframe1['DayType']
    PDP = dataframe1['PrevDayPercentage']
    PWP = dataframe1['PrevWeekPercentage']
    P2WP = dataframe1['Prev2WeeksPercentage']
    P12HP = dataframe1['Prev12HourPercentage']
    DayAfterPrevHoliday = dataframe1['DayAfterPrevHoliday']
    Summary = dataframe1['Summary']
    cs=[]
    timeArr =[]
    dayArr = []
    for elem in c:
        value = datetime.datetime.fromtimestamp(elem)
        str_time = value.strftime('%Y-%m-%d %H:%M:%S')
        dayArr.append(value.strftime("%A"))
        timeArr.append(str_time[11:])
        cs.append(str_time)
    return pd.DataFrame({"FreeSpacesDate":cs,"FreeSpacesPercentage":d,"Temperature":t,"Summary":Summary,
                        "DayType":dt,"PrevDayPercentage":PDP,"PrevWeekPercentage":PWP,"Prev2WeeksPercentage":P2WP,
                        "Prev12HourPercentage":P12HP,"DayAfterPrevHoliday":DayAfterPrevHoliday,"Time":timeArr,"Day":dayArr}
    # Execution logic goes here
    print('Input pandas.DataFrame #1:\r\n\r\n{0}'.format(dataframe1))
    return dataframe1,
```

Figure A.2: Execute Python Script component used for preparing the input features.