DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING, SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2021

Experimental Research on a Continuous Integrating pipeline with a Machine Learning approach

Master Thesis done in collaboration with Electronic Arts

SIGRÚN ARNA SIGURÐARDÓTTIR

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Abstract

Time-consuming code builds within the Continuous Integration pipeline is a common problem in today's software industry. With fast-evolving trends and technologies, Machine Learning has become a more popular approach to tackle and solve real problems within the software industry.

It has been shown to be successful to train Machine Learning models that can classify whether a code change is likely to be successful or fail during a code build. Reducing the time it takes to run code builds within the Continuous Integration pipeline can lead to higher productivity in software development, faster feedback for developers, and lower the cost of hardware resources used to run the builds.

To answer the research question: How accurate can success or failure in code build be predicted by using Machine Learning techniques on the historical data collection? The important factor is the historical data available and understanding the data. Thorough data analysis was conducted on the historical data and a data cleaning process to create a dataset suitable for feeding the Machine Learning models.

The dataset was imbalanced, favouring the successful builds, and to balance the dataset the SMOTE method was used to create synthetic samples. Binary classification and supervised learning comparison of four Machine Learning models were performed; Random Forest, Logistic Regression, Support Vector Machine, and Neural Network. The performance metrics used to measure the performance of the models were recall, precision, specificity, f1-score, ROC curve, and AUC score. To reduce the dimensionality of the features the PCA method was used. The outcome of the Machine Learning models revealed that historical data can be used to accurately predict if a code change will result in a code build success or failure.

Keywords

Machine Learning, Continuous Integration, Builds, Prediction, dataset, Code Change, Data Analysis

Sammanfattning

Den tidskrävande koden bygger inom pipeline för kontinuerlig integration är en vanlig faktor i dagens mjukvaruindustri. Med trender och teknologier som utvecklas snabbt har maskininlärning blivit ett mer populärt tillvägagångssätt för att ta itu med och lösa verkliga problem inom programvaruindustrin.

Det har visat sig vara framgångsrikt att träna maskininlärningsmodeller som kan klassificeras om en kodändring sannolikt kommer att lyckas eller misslyckas under en kodbyggnad. Genom att förbättra och minska den tid det tar att köra kodbyggnader i den kontinuerliga integrationsrörledningen kan det leda till högre produktivitet inom mjukvaruutveckling och snabbare feedback för utvecklare.

För att svara på forskningsfrågan: Hur korrekt kan förutsäga framgång eller misslyckande i kodbyggnad med hjälp av Machine Learning-tekniker för historisk datainsamling? Den viktiga faktorn är den tillgängliga historiska informationen och förståelsen för data. Noggrann dataanalys utfördes på historiska data och en datarengöringsprocess för att skapa en datamängd lämplig för matning av maskininlärningsmodellerna.

Datauppsättningen var obalanserad och för att balansera användes uppsättningen SMOTE-metoden. Med binär klassificering och övervakad inlärningsjämförelse gjordes fyra maskininlärningsmodeller, Random Forest, Logistic Regression, Support Vector Machine och Neural Network. Prestandamätvärdena som används för att mäta prestandan hos modellerna är återkallelse, precision, f1-poäng och genomsnittlig ROC AUC-poäng. För att minska dimensionaliteten hos funktionerna användes PCA-metoden. Resultatet av modellerna avslöjar att de med god noggrannhet kan klassificeras om en kodändring misslyckas eller lyckas baserat på den datamängd som skapats från historiska data som används för att träna modellerna.

Nyckelord

Maskininlärning, Kontinuerlig Integration, Byggnader, Förutsägelse, Datamängd, Kodändring, Dataanalys

Acknowledgments

First, I would like to thank my loving husband, Vignir Jónsson, for always believing in me and supporting my dreams. I also like to thank my wonderful kids for their patient and understanding. Super thankful for my family and friends for all their love and support. One of my dearest friend, Aldís Eva Friðriksdóttir, for her emotional support throughout my studies, and this project and for believing in me. I'm also super thankful to my supervisor, Fredrik Kilander, and my examiner, Anders Västberg, for their support, guidance, and constructive feedback. Then last but not least my great mentor and supervisor at Electronic Arts, Víðir Reynisson, this great project would not have been possible without him.

Stockholm, June 2021
Sigrún Arna Sigurðardóttir

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the key objectives for any software company is to make sure their software developers are as productive as possible, as it can be the difference between the success and failure of a company. Developer productivity can be negatively affected in multiple ways, for example, interruption from co-workers, multitasking, or losing focus while having to wait for some process to finish.

For most software companies code builds usually take a couple of minutes to run, however, for gaming companies, this process can be more time-consuming as large studio games can have a large monolithic code-base and the build artifacts can be around 50-100GB in size.

There are several ways to mitigate and help developers to stay productive. In this thesis, the focus was set to optimize the code builds by using Machine Learning to predict whether a software code build is likely to be successful or not. This prediction can be used in several ways, such as giving developers early feedback on whether their code build will be successful or not, prioritizing code builds in a build queue, or improving utilization of limited available hardware resources.

To be able to make this prediction a historical dataset of code builds and code changes were gathered from two tools, Jenkins and Perforce. A comprehensive data analysis and data cleansing the dataset was performed to identify and evaluate the possibility of implementing a solution that could give valuable and accurate predictions.

1.1 Background

Continuous Integration and Continuous Delivery/Deployment (also known as CI/CD) is a highly used term within software development that describes the process of building and delivering software. Continuous Integration (CI) is an automated process for software developers. Included in this process are automated builds, automated tests for the code change, and usually merging the code to a shared repository. Continuous Delivery is an automated process of deploying software to test environments, however, there is a manual step to deploy to the production environment. Continuous Deployment is an automated process of deploying software to production as effortlessly as possible. [1].

It can be quite challenging to implement the perfect CI/CD pipeline. For a gaming company developing a large studio games, it can be even more of a challenge as they typically have many developers contributing to a large shared monolithic code repository. A monolithic code repository is one big repository that stores the entire source code for a product or a project [2]. As a result of that, gaming companies will usually have a large codebase with a high volume of daily code changes (here referred to as code changes, new lines of code to the source code, and as well dependency changes) and possibly many builds to execute as they will need to build large software artifacts and possibly support many gaming platforms.

This work was done in collaboration with a gaming company, Electronic Arts, that will be called the host company for the purpose of this thesis. The host company develops large studio games. Their development, testing, and release workflows are driven by a CI/CD farm (a farm is a group of servers) that continuously builds, tests, and merges code to the mainline branch (here referring to the main branch of development).

This company can have up to 500 active developers working in the same monolithic code repository with a large volume of code changes or up to 680 changes a week. This can result in a broken mainline that has a severe and negative impact on the development teams. The company leverages various methods to optimize for a stable, functional mainline at all times, such as parallel pre-submission validation of changes but there is always room for improvement.

One possible source of improvement was the historical build data that the company has and has been gathering for the last four years. This data includes information about historical builds and whether they have failed or succeeded. With this work, research has been conducted where Machine Learning methods have been evaluated for improvements on the CI pipeline.

1.2 Problem

To frame the problem for this work, the author needed to understand and narrow down the objective and the outcome. The host company was in the process of making improvements to its CI/CD pipeline. When reviewing the current pipeline at the time, it was clear that the company was using the latest technology and trends when it comes to software practices.

The objective of the work was at first very high level, and the scope needed to be reduced to fit the timeline and target dates. Thus the scope was narrowed down to the CI part of the pipeline and focused on the code builds. The reason for this was that the code builds are one of the most time-consuming parts in the process and when a code build fails after a developer has been waiting for it to finish, they need to continue working on the code that failed the build to fix it and submit the code change to be built again. This time-consuming process can affect the developer's productivity.

After the scope was reduced the objective became more clear: how can one predict if a code change is likely to fail during a software build?

Another reason for solving this problem was the data. The host company had been gathering build data for almost four years, and this sounded very promising to be a good starting point from which to implement a Machine Learning model.

After further research and analysis on the code build dataset, it became clear that there was some crucial information missing before it was possible to implement a successful Machine Learning model. The information that was missing from the code build dataset was detailed information such as how many lines or files changed, and how many commits were made for this code change. From that understanding, it became clear that additional data needed to be fetched from the company's version control system, Perforce. Perforce is a version control system that is preferably used in the enterprise software industry[3]. The data from Perforce had to be correlated with the build data previously gathered to create a complete and useful dataset.

To address the problem this research question was the guidance throughout the work:

- How accurate can success or failure in a code build be predicted by using Machine Learning techniques on the historical data collection?

1.3 Purpose

The purpose of this work was to discover the potential of the company's historical data to understand if a Machine Learning models could be implemented to accurately predict if a code change is likely to fail or be successful without actually running the code build.

1.4 Goal

The outcome of this work will be used as a crucial input into the company's continued decision-making process moving forward, and will help steer them towards more stable builds. Additionally, if possible, a functional prototype in software can help prove the merit of this approach within the software development industry. In the case where the evaluation of the companies pipelines, metrics, and data sets has shown to be severely lacking then a step back would have been taken and a focus on how one can be prepared by having gathered the right insights. Then understand where this went wrong before, and hopefully build a more hypothetical proof-of-concept to show the value-add of the additional information is gathered and how it can be leveraged for improved and autonomous decision-making later on.

1.5 Ethics and Sustainability

From an ethics point of view the outcome of the project might give the possibilities of automation in some parts of the CI pipeline. The disadvantage of that would be that it might eliminate some jobs at software companies. The advantage would be that if a part of the CI pipeline is automated that could lead to developers producing fewer bugs with their code and therefore improving code quality and allowing the developers to be more productive. Regarding sustainability, if the desired outcome for this project is successful that might lead to fewer resources needed for the data-centres which have a

positive impact on the environment as well could save software companies some money.

1.6 Research Methodology

A Quantitative Research method was followed throughout the work with a focus on the Experimental Research method. The raw dataset was investigated to implement a proof-of-concept prototype for improvements that can be applied to the CI pipeline[4]. After conducting both data cleaning and data analysis on the dataset and focusing on the quantitative variables available, it was possible to measure and validate that the data available might predict the likelihood of a successful build for a given code change. After a proper literature study was carried out a selection of algorithms was recommended for Machine Learning models. The literature study is discussed in more depth in chapter 2, section 2.3.

1.7 Delimitations

In a perfect world, it would have been optimal to be able to apply the model in the production pipeline to see how it can reduce both the build time and the need for resources. Due to time limits, this was not possible.

1.8 Structure of the thesis

Chapter 2 presents relevant background information about Code Builds, Continuous-Integration, Data Cleaning, and Data Analysis, Algorithms and Model Training, Binary Classification, Supervised Learning, Class Imbalance, Feature Engineering, and related work to the subject of the thesis.

Chapter 3 presents the methodology and method used to solve the problem, the environment used, how the dataset was created, and what method is used to evaluate and validate the results.

Chapter 4 present the implementation of the work carried out.

Chapter 5 presents the results and analysis of the experiment.

Chapter 6 present the conclusion and future work.

Chapter 2

Extended Background

This chapter provides background information about Continuous Integration workflow and pipelines in today's industry as well as for Electronic Arts in section 2.1, data cleaning and data analysis in section 2.2, and Machine Learning algorithm selection in section 2.3. The chapter also describes binary classification in section 2.4, supervised learning in section 2.5, class imbalance in section 2.6, feature engineering in section 2.7, and related work for using Machine Learning in the Continuous Integration pipeline in section 2.8.

2.1 Continuous Integration

2.1.1 Code builds

The automated process of building code transforms the source code into deliverable software. Jenkins is an open-source tool that can be used to automate code builds, tests, and deploying software[5]. Developers may use build systems to continually re-build testable artifacts after making code changes. Ghaleb et al.[6] conducted an empirical study in 2019 on the long duration of CI builds. Their study was conducted on 67 GitHub projects with 104,442 CI builds. The result shows that many software projects have build time that takes longer than expected or roughly around 10 minutes.

2.1.2 CI in today's industry

The CI workflow can be very similar between software companies. A classic workflow is when a developer creates a branch from the main branch. The mainline branch is the latest, up-to-date version of the code repository and

when creating a branch from the mainline branch a copy is created. The developer makes changes to the code working on the branch, and commits these changes on the branch. The branch is then pushed to the repository. A CI tool can be used to integrate into the system hosting the source code. Version control system, will then trigger code builds and possibly automated tests. If both builds and tests pass, or in other words the status of the CI pipeline is green, then the developer can merge the branch to the mainline branch. However, it is very common to have a review process before merging another branch to the main branch if the developer created a pull request for the branch. A pull request is a request for merging a change into the mainline branch and allows fellow developers to review the code change before it is merged into the mainline branch as well as give constructive feedback on the code change if needed. The main benefit of applying the CI software practice is to automate the process of delivering software as well improving the reliability and quality of the software[7].

2.1.3 CI at Electronic Arts

Today one can find many choices of tools to implement a CI practice. The host company, uses a tool called Jenkins (introduced in section 2.1.1) for their CI pipeline. Figure 2.1 represents an overview of the CI pipeline. The developer creates a code change locally, then uses the shelving tool that shelves the developers' changes to trigger the Jenkins jobs. The Jenkins jobs will then unshelve the changes and run the automated code builds. Jenkins then returns the results of the builds if they ran successfully or not. If the code build failed, logs from the build are available for the developer. From the information in the logs the developer can then identify the problem, change the code accordingly, and submit the updated code change.

The host company is an enterprise company where large engineering teams are working on big projects and pushing code to a monolithic repository. The average submissions to the mainline are up to 680 changes a week. What happens under the hood when the internal tool is used to start the Jenkins jobs is that it will first make sure that there are available build servers to run the builds. If there are no servers available then the code change is queued. The reason there might not be an available server to run the code build is that there are limited numbers of servers available. On these servers, the code builds are run in parallel before they are pushed to the mainline/Perforce (Perforce is introduced in section 1.2).

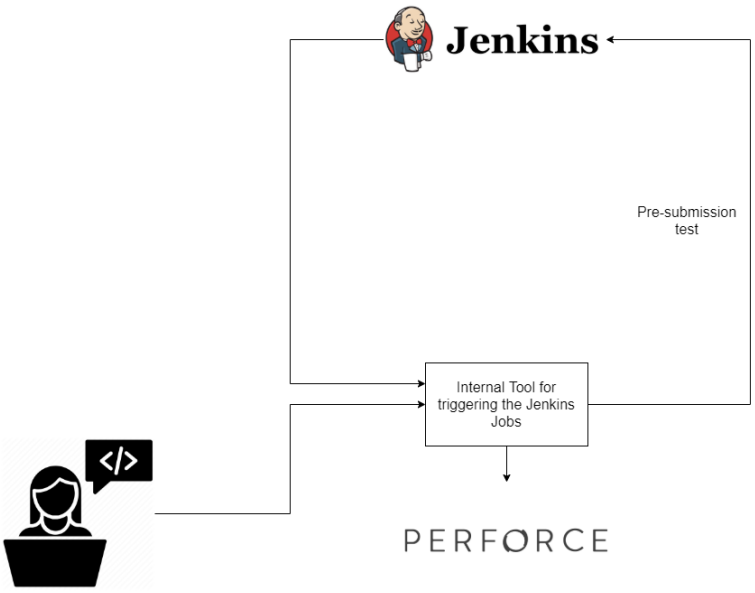

Figure 2.1: CI workflow at the host company. Developer creates a code change locally and shelves the changes using the internal tool/shelving tool to trigger the Jenkins jobs (that unshelve the code change) to run the code builds and pre-submission test and results are shown in the internal tool if the code build ran successfully or failed. If the code build ran successfully it's pushed to Perforce else the developer change the code and repeats the process.

A reason for failing code builds could be the human aspect and one needs to consider what preventive measures are done for implementing a more stable code. As examples of those preventive measures are code guidelines and code standards. Another preventive measure is a code review. The purpose of code reviews is to have other developers reviewing a code change before it is merged with the main branch. This is done by creating a pull request. The developers that are set as reviewers on pull requests can give constructive feedback in regards to best practices and code guidelines.

2.2 Data Cleaning and Data Analysis

The data cleaning process is a crucial first step before diving into the data analysis process[8]. One first needs to get insight and understanding of the raw data. The main steps included in the data cleaning process are tidying the data, detecting outliers, and diagnosing the data[10]. The raw data is likely to be noisy, incomplete, inconsistent, and difficult to analyze. Cleaned data

stands a greater chance of being analyzed successfully[8].

The value of carrying out a thorough data analysis is that it allows the usage of the dataset for implementing a Machine Learning model that can give valuable results.

The raw dataset was from the hosting company that had collected historical build logs for roughly 4 years and revision history for longer period of time. A major part of the work was conducting a comprehensive data analysis on both the build logs and the revision history. The build logs came from Jenkins but were stored in a tool called ElasticSearch[9]. The revision history data was from a tool called Perforce. The data from Perforce consisted of detailed information regarding each code change, for example number of lines added, removed, or changed. A more detailed description of the data is discussed in chapter 3, section 3.4 The data that was most interesting in the historical build logs was the overall results of failed and successful builds. This information was useful as they were represented as labels in the Machine Learning model. Then these values were correlated with the data from Perforce for feature purposes for the Machine Learning model. To be able to correlate this data a python script was implemented. The next chapter (chapter 3) gives a more thorough description of the process of both fetching the data, creating the dataset, and analyzing the dataset. As well, chapter 3, presents a visualization of the data and the findings for feature selection for the Machine Learning model.

2.3 Algorithms

A comparison of four Machine Learning algorithms was done. Those algorithms are Random Forest, Logistic Regression, Support Vector Machine, and Neural Network.

Selecting Random Forest and Logistic Regression was based on the literature study that was carried out and is discussed in the section 2.8 Related Work. For the sake of the experiment Support Vector Machine and Neural Network were also selected for comparison against Random Forest and Logistic Regression.

2.3.1 Random Forest

The Random Forest is a collection of decision trees and is a *bagging ensemble* method [10]. The growing trees add a layer of randomness in the Random Forest algorithm by looking for the best feature in a subset of the random

features rather than the best feature overall when splitting a node. More tree diversity is produced with the Random Forest Algorithm than with the Decision Tree algorithm [11]. When used for classification problems, the outcome from a Random Forest algorithm is a summed up majority vote. All of the trees are trained differently as each tree receives its unique collection of features. The sheer quantity and variety of trees increase the overall model's resilience and tolerance to overfitting while also allowing it to capture more intricate relationships[12].

2.3.2 Logistic Regression

The Logistic Regression or logistic function, which is also known as the Sigmoid function, explains the characteristics of economic growth in biodiversity. This function explains how these characteristics rise rapidly and eventually reach the environment's carrying capacity[13]. The likelihood that an object belongs to a given class, for example, the likelihood of an email is spam or not, is typically estimated by using the Logistic Regression algorithm. The weighted sum of the input characteristics (plus a bias term) is computed by the logistic regression model, which then outputs the logistic of this result[14].

2.3.3 Support Vector Machines

The Support Vector Machines [15] (SVMs) algorithm is a concept of maximum margin classifiers. SVM is a technically sound approach to model complexity management. The great potential of SVM comes with balanced accuracy and reproducibility by learning data classification patterns. SVM uses a hyperplane for decisions to classify data points to a label. Identifying a consistent hyperplane that maximizes the distance/margin between the support vectors of two class labels is the process of training an SVM classification function [16].

2.3.4 Neural Network

A Neural Network [17] is a mathematical model of biological neurons that has both input and output units. The units in a neural network are connected forming a graph. These units play the role of the input unit, output unit, and computing logic. Processing modules, nodes, and neurons are all terms used to describe these units. The units are bound by arcs that are either unidirectional or bidirectional, with weights indicating the frequency of the connections. Neural networks have recently resurfaced as a viable alternative to a variety

of traditional classification systems. When it comes to tackling real-world problems Neural Network has a strong practical effects as well when it comes to theoretical basis. [17]

2.4 Binary Classification

Binary classification is a task that involves two class labels. In this work one class for a state that is abnormal and the other class for a state that is normal. Then these states are assigned to the binary values 1 and 0, for a normal state would be 1 and for an abnormal state 0[18]. In this work, the class for the normal state represented the successful build and then the class for the abnormal state represented the failed build.

2.5 Supervised Learning

Supervised learning is a process when an algorithm learns from a pre-classified training dataset. The goal of supervised learning is to train an algorithm to define correlations between the variables being trained and the labels. When the algorithm has accomplished a desirable performance the learning process stops. Supervised learning is commonly used when implementing Random Forest, Logistic Regression, Support Vector Machines, and Neural Network[19].

2.6 Class Imbalance

When the data is heavily biased in favour of one class, the dataset is imbalanced. This will result in a naive Machine Learning model that classifies nearly all samples as belonging to the over-represented class and maximizing the accuracy score [20]. The dataset produced for this experiment had class imbalance favouring the successful builds. Numerous approaches can be taken to address the issue of class imbalance. One of those approaches that were used in this experiment is Synthetic Minority Over-sampling Technique (SMOTE) [21]. SMOTE is an oversampling method proposed by Chawla et al. [22]. This method over-samples the minority class by creating synthetic samples. By using SMOTE the accuracy of the minority class can be improved.

2.7 Feature Engineering

Feature engineering is one of the most important parts of implementing a successful Machine Learning model [14]. This process, feature engineering, consists of doing feature selection, feature extraction, as well collecting new data and create new features. The feature selection process includes selecting features that are most useful when it comes to training the models. The feature extraction process includes reducing the dimensionality of the feature set, this can be done by combining features that have a high correlation [14].

2.7.1 Principal Component Analysis

The Principal Component Analysis (PCA) algorithm was used for this work to reduce the dimensionality of the feature set. The goal of PCA is to find the features that have the least correlations and extracts the most important information from the dataset with fewer features.

When working with classification models PCA can improve the performance. PCA comes with the Scikit-Learn library and can easily be applied. The PCA function takes one parameter that is the number of components or features the function should extract from the dataset. [23].

How PCA works is that the dataset is transformed into a new space by PCA and each of the matrix's new column vector is orthogonal. The covariance matrix of the data is converted into column vectors that can describe particular percentages of the variation using PCA. Using PCA can be beneficial when working with high-dimensional dataset as they tend to overfit the Machine Learning models[24].

2.8 Related Work

Ananthanarayanan et al.[25] introduced a change management system at Uber that is called SubmitQueue. SubmitQueue consists of three modules: there is the speculation engine, the conflict graph, and then the planner engine. This system has shown to be successful at keeping the mainline of deployment stable or in other words the failure of builds was drastically reduced after the introduction of SubmitQueue. This system is very complex but yet an interesting approach to resolving a common problem within the software industry of deploying software. The speculation engine is a Machine Learning model implemented with a probabilistic model that is powered by logistic regression that has a 97% accuracy.

The conflict graph compares two code changes and if they affect the same target/code they are likely to conflict by using the hash target algorithm. The speculation engine uses the conflict graph to be more accurate in speculating what code change is likely to be successful. The result from the speculation engine is then sent to the planner engine which then schedules the builds that will be executed from the information from the speculation engine.

The dataset that the conflict graph and the speculation engine are trained on is the historical data of build logs and commit logs from Uber. This change management system can handle thousands of commits every day to massive monolithic repositories.

SubmitQueue gives a great example of the possibilities that can be achieved if you have the data available as well as gives a good idea of the methods that can be applied for conducting this kind of experiment. With this project, a similar approach was used for the speculation engine, by implementing one of the modules within the change management system that can output accurate predictions.

Hassan et al.[26] conducted an empirical study to implement a build prediction model to predict if a build for a code change is likely to either fail or be successful before the build is performed. The writers used a TravisTorrent dataset and within that dataset are build logs from the build systems Ant, Maven, and Gradle. Hassan et al. focused on investigating the need for, and feasibility, of a change-aware build prediction model by looking at the software build execution time, commit, and consecutive build status change.

The results from this experiment were shown to be successful. By applying the Random Forest algorithm to predict if a code change will be successful, achieves an outcome prediction of 87% F-Measure on the build systems mentioned above for cross-project build-outcome prediction.

Saidani et al. [27] carried out an empirical study by proposing a novel search-based approach based on Multi-Objective Genetic Programming. The dataset that was used for this study was from long-living projects from a Travis CI build system. The focus on the attributes in the dataset was the team size, calculations of metrics, and changes of files.

The experimental results of their study show that their solution outperforms other Machine Learning approaches to the same problem as well covers the imbalanced dataset aspect.

2.9 Summary

To summarize the extended background chapter, the main area for this thesis is the workflow of a CI process, both how it is described in general and how it works at the host company. High-level description of the data cleaning and data analysis was presented. The Machine Learning algorithms used for the experiment as well as describing the framework for the models, imbalance data and feature engineering were introduced. From the literature study, one can see that the same problem has been tackled in the software industry with great success but with different approaches. However, one must consider that each workflow, pipeline, and historical data for a software company differentiate from one another, and with that in mind, the related work was used as an important input for this work.

Chapter 3

Methodology

This chapter provides an overview of the research method used in this work. Section 3.1 describes the research process. Section 3.2 details the research paradigm. Section 3.4 focuses on the data cleaning and data analysis process used for this research. Section 3.3 describes the environment used for the research, both hardware and software. Section 3.5 explains the method used to evaluate the validity and performance of the Machine Learning methods.

3.1 Research Process

One of the research methods that was carried out for this work was a quantitative data analysis by conducting exploratory data analysis on the dataset. The results were visualized for explaining and understanding the findings from the data analysis.

The process of cleaning and analyzing the data included removing outliers from the dataset as well creating data points that are represented as features and labels that were used as input for the models. The data points that were created are based on the literature study. The output of this process was a dataset that was suitable for the implementation of a Machine Learning model with supervised learning and binary classification.

The research process that was used for the Machine Learning aspect was the experimental research method by doing a comparison of four algorithms, Random Forest, Logistic Regression, Support Vector Machine, and Neural Network.

The end goal was to have a proof-of-concept Machine Learning model with high accuracy of predicting true negatives and true positives. The performance metrics used to measure the accuracy of the models are discussed in section 3.5. Then a small experiment was done where the prediction method was used with the best performing Machine Learning model showing results for a duration of prediction of a code change is shown in chapter 5.

3.2 Research Paradigm

The framework that was used is the positivists' paradigm, as the research methodology that was used was quantitative methods. The problem that was solved was a real problem of predicting if a code change will fail or be successful before it goes through the build process. To solve this problem experimental research was carried out by implementing four Machine Learning algorithms using part of the dataset for the training dataset and the test dataset and a comparison was done on the results from the algorithms[28].

3.3 Environment

The experiment for this work was done on a Dell Precision 5820, Intel Core i9-10980XE CPU @ 3.00GHz 3.00 GHz, 128 RAM, 64-bit OS, x64-based processor, with Windows 10 enterprise. All the code was written in Python version 3.8.5[29]. The Python scripts to gather and parse the data were implemented in Visual Studio Code, version 1.55.0. A Jupyter notebook version 6.1.4[30], was used for the data analysis and the implementation of the Machine Learning models. The libraries that were used for the data cleaning and data analysis part was JSON encoder and decoder version 2.0.9[31], NumPy version 1.19.2[32], Matplotlib version 3.3.2[33], Seaborn version 0.11.0[34], and Pandas version 1.1.3[35]. The libraries that were used for the Machine Learning part was Scikit-Learn version 0.24.1[36] and, Imbalanced-Learn version 0.8.0[37].

3.4 Data Cleaning and Data Analysis process

A Python script was implemented to fetch the data, the output from that script was a raw dataset in a JSON format. To clean up the raw dataset another Python script was implemented to parse the JSON output giving data points that could be used as input for the Machine Learning models.

3.4.1 Fetching the data

Historical data was fetched from two sources, Perforce and ElasticSearch. The data from ElasticSearch was from three separate ElasticSearch instances and the data period was from 1st of January, 2016 until 1st of April 2021. The data from these two sources were joined together on a unique variable, an ID, that was available in both Perforce data and ElasticSearch data. The output from this script of fetching the data was a raw JSON file consisting of only successful and failed builds. Cancelled builds were removed from the raw data. Important data, more descriptive information about the code change, for each build comes from Perforce in a raw text free format.

3.4.2 Parsing the data

After fetching the raw data another Python script was implemented to parse it. The raw data contained important information regarding historical code changes in a text format. Variables were created to keep a count on the statistical changes performed in a code change. Based on the literature study, discussed in chapter 2, section 2.8, the important data points for feature selection are the following; number of lines changed, number of lines added, number of lines removed, number of files changed, number of commits for each change, number of authors, and feature for each file extensions.

In listing 3.1 is an example of a JSON object created after processing the raw data (this object does not show a complete example of all the file extensions in the dataset). The size of the dataset after parsing the raw data is 5.12MB or 8746 builds. This dataset includes 266 columns where 265 are features that are used to train the models. The features have statistics about the code change made as well a binary value, buildSuccess, that represents if the build was successful or not. If the value for buildSuccess is 1 then the build was successful, and 0 if the build failed.

Listing 3.1: Example of a JSON object after parsing the raw dataset and creating features for the purpose of training the Machine Learning models.

```
{
        "json": 2,
        "cpp": 25,
        "py": 2,
        "dll": 2,
        "pdb": 2,
        "dbx": 7,
        "adf": 1,
        "exe": 1,
        "cs": 4,
        "xml": 1,
        "h": 20,
        "build": 3,
        "ddf": 4,
        "numberOfCommits": 15,
        "numberOfAuthors": 12,
        "numberOfFilesChanged": 74,
        "numberOfLinesAdded": 42,
        "numberOfLinesDeleted": 5,
        "numberOfLinesChanged": 152,
        "numberOfChunksAdded": 12,
        "numberOfChunksDeleted": 2,
        "numberOfChunksChanged": 34,
        "buildSuccess": 1
}
```

3.4.3 Imbalanced dataset

After thorough data analysis, an understanding of the data was clear which revealed that the dataset was imbalanced, as shown in figure 3.1. The majority of the historical code builds had run successfully, and where 86% of the total data included successful builds. Working with an imbalanced dataset trying to train a Machine Learning model can be difficult as the model will have problems distinguishing the minority class from the majority class. In this case the minority class was the failed builds with 14% of the total dataset.

To balance out the dataset the SMOTE method was used to create a balanced dataset, or 50% successful builds and 50% failed builds. SMOTE is only used on the training data after the split on the dataset has been done. In chapter 5 results are shown from the models both without applying SMOTE on the training data and applying SMOTE on the training data.

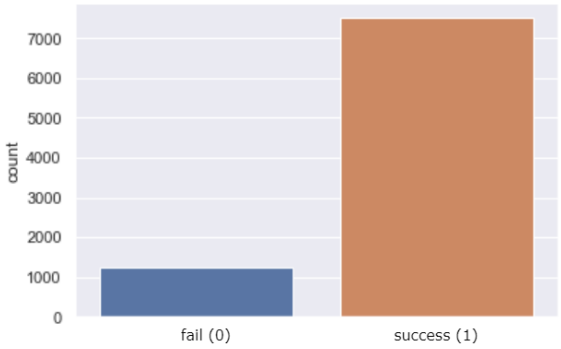

Figure 3.1: Count plot visualizing the imbalance in the dataset where 1236 are failed builds(0) and 7510 are successful builds(1).

3.5 Method Evaluation

One of the aspect of building an efficient Machine Learning models is understanding how to evaluate the performance of the models. This gives a good framework for when both implementing the models and when analysing the results. chapter 1. Hassan et al.[26] used confusion matrix, precision, recall, F1-score, and ROC area to evaluate the predictions from the model they implemented. As well as these metrics are popular when it comes to evaluate classification models[38].

3.5.1 Metrics and Performance

With a focus on correctly classified binary labels, a confusion matrix was one of the classification metrics used for measuring the performance of the predictions of the models. As seen in figure 3.2 this matrix can give four outcomes, true positive (TP), true negatives (TN), false positives (FP), and false negatives (FN) [39]. The confusion matrix makes it is easy to visualize the distinction between the classes and how a Machine Learning model is able to correctly predict the class labels.

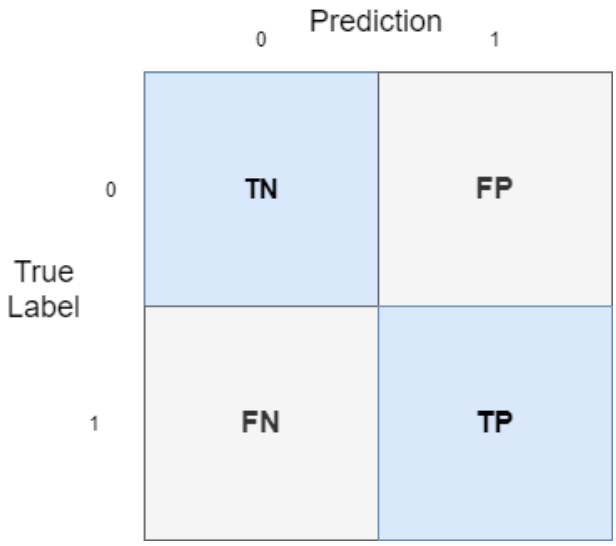

Figure 3.2: Confusion matrix has the following outcomes: true positive(TP), true negative(TN), false positive(FP), and false negative(FN). [39]

Precision, recall, F1-score, specificity, ROC curve, and AUC score are useful metrics for evaluating supervised classification models[38]. These metrics are all connected to one another and are generated from the confusion matrix using the true positives, true negatives, false positives, and false negatives.

The percentage of expected positive cases that are correctly real positives is defined as precision[40].

$$precision = TP/(TP + FP) \text{ [40]} \tag{3.1}$$

The percentage of real positive cases that are correctly predicted positive is defined as recall[40].

$$recall = TP/(TP + FN) \text{ [40]} \tag{3.2}$$

Recall and precision is used to calculate the F1-score. This metric is the harmonic mean of the recall and precision metrics.[40].

$$F1 = 2 * (recall * precision)/(recall + precision) \text{ [40]} \tag{3.3}$$

Specificity is the opposite of the recall metric and identifies all the percentage of real negative cases that are correctly predicted as negative[41].

$$specificity = TN/(TN + FP) \text{ [41]} \tag{3.4}$$

ROC, or Receiver Operating Characteristic is a visualization that plots the true positive rate against the false positive rate using the prediction probability. A visualization of a ROC curve is beneficial when evaluating a classifier and understand if the classifier is performing well in distinguishing between classes. AUC, or area under the ROC curve, summarizes the ROC curve to one value that reflects the classifier's predicted performance[42].

3.5.2 Model Validation

For the training and validation, the k-fold cross-validation method was used where the training data is randomly split into k-folds. For model training, k - 1 folds are used and for performance assessment, one fold is used. This process is then executed k times to get k models as well the performance assessment[43]. Where k for this work was set to 10. Ron Kohavi[44] experiment shows when using real-world dataset with 10 folds cross-validation is the optimal value for k even if even if there's room for more computational capacity for a higher k folds.

3.5.3 Baseline

A dummy classifier is used to set a baseline to compare how the other models are performing. Dummy classifier gives intentionally bad outcomes and the purpose of using them is not to actually make predictions. The dummy classifier has a parameter called `strategy`. The `strategy` parameter can be set to `uniform`, `most_frequent`, or `stratified`. The `uniform` classifier predicts if a incident is related to which class. The `most_frequent` classifier as the name indicates predicts the most common label. The `stratified` classifier will maintain a ratio with its estimation based of the class distribution in the training data[45].

Chapter 4

Implementation

This chapter describes the experimental iterative design process in section 4.1, data preparation and pre-processing of the dataset in section 4.2, how PCA was implemented in section 4.3, the implementation of the four Machine Learning models in section 4.4, and how the SMOTE method was used on the imbalanced dataset in section 4.5.

4.1 Experimental iterative design process

To refine the scientific instrument until it was fully tuned the experimental iterative design process was used to continuously improve the models. The starting point was implementing raw and basic Machine Learning models (with no hyperparameters) and few features, then step-by-step tweaking one thing at a time and adding more features to the dataset as tuning the hyper-parameters until sufficient results were achieved. Figure 4.1 gives an overview of the workflow used for the experiment.

4.2 Data preparation and pre-processing

Before splitting the data all null values were removed and 0 integers were set instead as a value. The dataset consisted of 8746 builds, where 7510 were successful builds and 1236 were failed builds. The features were set to variable X and the labels were set to variable y. Then the `train_test_split` method from Scikit-learn was used to split the features and labels into a training set and a test set. The split for all of the models was 75% training set and 25% test set. The random state was set to 42 for all the models as well, to keep consistency of the random split of the data. For the validation set,

Figure 4.1: Workflow of the experiment

the cross-validation method was used with 10-k folds on the training data to keep the test data isolated and avoiding data leakage. Data leakage is when information from the training set leaks to the test set. Before training the models the values in the training set was scaled using StandardScaler from the Scikit-Learn library. The StandardScaler calculates a standard deviation and means for each column [46].

4.3 Principal Component Analysis

For feature engineering the PCA method was applied on the training set, after it was scaled with StandardScaler, to reduce the dimensionality of the features. Before selecting the value for `n_component` a graph was plotted explaining the cumulative explained variance ratio. In figure 4.2, the curve shows how the 265 components (number of features) can be found in the cumulative explained variance. The 125 first components contain around 90% of the variance.

Selecting more than 125 components might risk the curse of dimensionality and could overfit the models. Therefore it was concluded from the graph that 125 components were selected for the implementation of PCA.

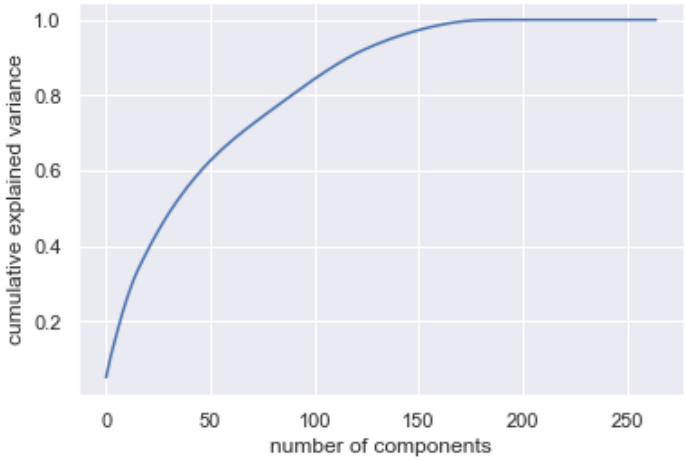

Figure 4.2: Principal Component Analysis

4.4 Classification Models

A pipeline, `sklearn.pipeline` from the Scikit-Learn library, was implemented for each of the Machine Learning models to avoid data leaking from the training set to the test set.

4.4.1 Grid Search

To find the best hyperparameters to tune the models the grid search[47] method was used. Scikit-learn library has the GridSearchCV method available. For each model a parameter grid was implemented with a range of values per parameter then the GridSearchCV evaluates each of the values. After evaluating all values for the hyperparameters the grid search returns the best hyperparameters values to tune each model[38].

4.4.2 Cross-Validation

The Scikit-learn k-fold cross-validation was used to implement the cross-validation. Scikit-learn uses the stratified k-fold cross-validation for classification problems. The stratified approach maintains the ratio of values for

each class and per folds[48]. As mentioned in chapter 3, section 3.5, the k was set to 10 and the cross-validation only done one the training set to keep the test set separate to perform the predictions for the performance metrics.

4.4.3 Dummy Baseline Model

A dummy model was implemented to create a baseline. For this model, the DummyClassifier from Scikit-Learn was used. The model is very basic and has one parameter, `strategy` which for this experiment was set to stratified. The results from the dummy baseline model consist of the same performance metrics mentioned in chapter 3, section 3.5. The baseline is used for comparing the results from the Random Forest, Logistic Regression, Support Vector Machine and Neural Network models. This gives an understanding and validates the results from the other models if they are performing well or not.

4.4.4 Random Forest

There were several hyperparameters to tune to optimize the performance of Random Forest.The result of GridSearchCV to find the best hyperparameters to tune the Random Forest model is shown in table 4.1. The `max_depth` parameter sets the depth per each tree. The `max_features` parameter sets the number of features that are used for constructing a tree and can help the model in learning distinct data from the training set.
The `min_samples_split` parameter sets the smallest amount of data points that are needed to separate a tree node. The `n_estimators` parameter set the number of trees that should be created for the Random Forest model[49].

Table 4.1: Hyperparameters used to tune the Random Forest Classification model

Random Forest Hyperparameters	
Hyperparameter	Value
bootstrap	True
max_depth	16
max_features	auto
min_samples_split	2
n_estimators	56

4.4.5 Support Vector Machine

The name of the hyperparameters and the values that were used to tune the Support Vector Machine model are shown in table 4.2. The `kernel` parameter set was the RBF (radial basis function). The `C` parameter regulates the hardness of the margin. The `gamma` parameter is set to auto, this parameter sets the size of the kernel[50].

Table 4.2: Hyperparameters used to tune the Support Vector Machine Classification model

Support Vector Machine Hyperparameters	
Hyperparameter	Value
kernel	rbf
C	10
gamma	auto

4.4.6 Logistic Regression

In table 4.3 are the names of the hyperparameters used to tune the Logistic Regression model and the values set for each hyperparameter. The `C` parameter

sets the weight of the penalty for the model. Then the `penalty` parameter which is by default l2, which is also known as ridge regression, by summarizing the squares of the coefficients to the cost function the `l2` punishes the high coefficients. The `solver` parameter is set to `sag`, also known as the stochastic average gradient [51].

Table 4.3: Hyperparameters used to tune the Logistic Regression Classification model

Logistic Regression Hyperparameters	
Hyperparameter	Value
C	0.0001
penalty	l2
solver	sag

4.4.7 Neural Network

In table 4.4 are the names of the hyperparameters used to tune the Neural Network model and the values set for each hyperparameter. The `alpha` parameter penalizes weights with high magnitudes which helps to minimize overfitting. The `hidden_layer_sizes` parameter sets the number of neurons in the hidden layer in the Neural Network model. The `solver` parameter set for optimizing the weight was `lbfgs`. The `relu` activation function was used which is the default activation function[52].

Table 4.4: Hyperparameters used to tune the Neural Network Classification model

Neural Network Hyperparameters	
Hyperparameter	Value
alpha	0.1
hidden_layer_sizes	5
max_iter	5000
solver	lbfgs

4.5 SMOTE

SMOTE was only used on the training set, `X_train` and `y_train`. The SMOTE method from Imbalanced Learn[53] library was used to over sample the training set using the `fit_resample(X_train, y_train)`. As seen in figure 4.3, this resulted in having in total 11308 builds, 5654 that were successful builds and 5654 that were failed builds.

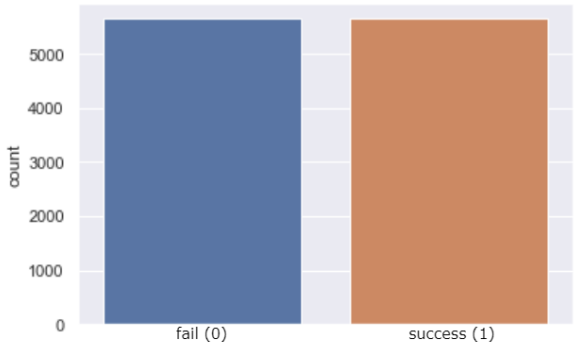

Figure 4.3: Count plot visualizing the training set after applying SMOTE on the training set to balance it, resulting in 5654 failed builds(0) and 5654 successful builds(1).

Chapter 5

Results and Analysis

This chapter presents the results from the experiment of training Random Forest, Logistic Regression, Support Vector Machine, and Neural Network with the training dataset and results of performance metrics by evaluating the model with the test dataset. As well as the results from using SMOTE on the training dataset. The goal of this project was to implement a proof-of-concept Machine Learning models to answer the research question: How accurate can success or failure in code build be predicted by using Machine Learning techniques on the historical data collection? To answer the question, a comparison has been done on the models to reveal the best performing model and the accuracy of the models by focusing on the weighted F1-score, recall, specificity, precision, ROC curve, and AUC score.

5.1 Confusion Matrix

The following confusion matrices reveal the classified results from the Dummy baseline, Random Forest, Logistic Regression, Support Vector Machine, and Neural Network. For comparison, the results from the Dummy baseline model sets the baseline for the other models. This means that if true positives and true negatives are lower in the models than the baseline then those models are not able to classify with good accuracy. The figures show the difference between using SMOTE, tuned hyperparameters with GridSearchCV, and dimensionality reduction with PCA and then without using SMOTE to balance the dataset.

5.1.1 Dummy Baseline model Confusion Matrix

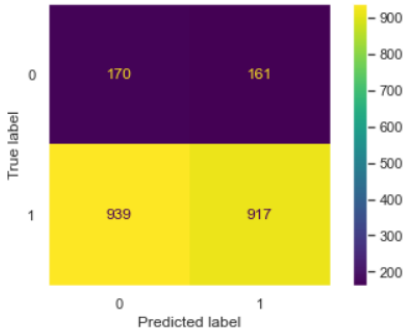

Figure 5.1: Dummy Baseline Classifier Confusion Matrix with SMOTE

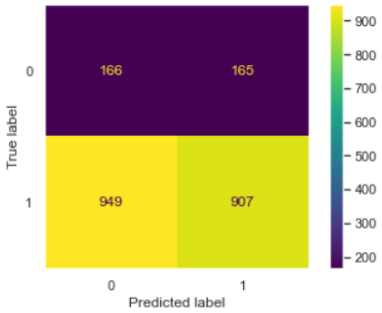

Figure 5.2: Dummy Baseline Classifier Confusion Matrix without SMOTE

The results from the Dummy baseline models are from using SMOTE and without using SMOTE on the training set. Figure 5.1 shows the confusion matrix after applying the SMOTE method on the training set, where the Dummy baseline model correctly predicted 170 true negatives, failed builds, and 917 true positives, successful builds. Figure 5.2 shows a confusion matrix without applying the SMOTE method, the model was able to predict correctly failed builds is 166 builds, the true negative, and correctly predict the successful builds is 907, the true positive. In both matrices the recall and specificity is around 50%, which indicates that there is a 50% probability that a value is classified as either true negative or true positive.

5.1.2 Random Forest Classifier Confusion Matrix

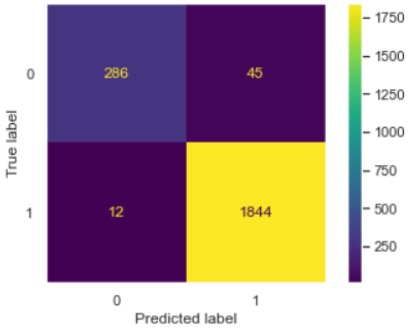

Figure 5.3: Random Forest Classifier Confusion Matrix with SMOTE

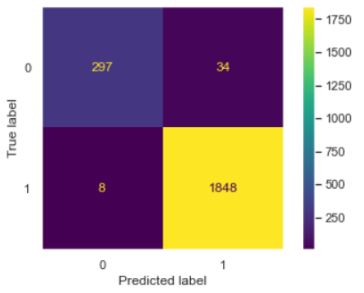

Figure 5.4: Random Forest Classifier Confusion Matrix without SMOTE

The results from the Random Forest Classifier, in figures 5.3 and 5.4, shows that the model can correctly predict 286 true negatives and 1844 true positives after balancing the training data with SMOTE. Without balancing the dataset with SMOTE the model was able to correctly predict 297 true negatives and 1848 true positives. For comparison the recall metric when using SMOTE is 99.3% and without SMOTE it is 99.5%, then the specificity metric when using SMOTE is 86.4% and without SMOTE it is 89.7%.

5.1.3 Logistic Regression Classifier Confusion Matrix

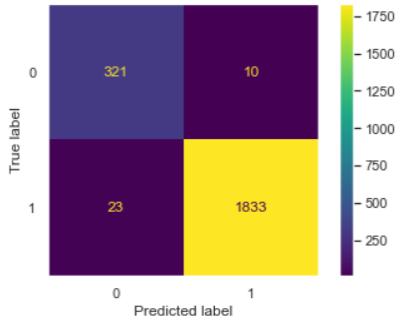

Figure 5.5: Logistic Regression Classifier Confusion Matrix with SMOTE

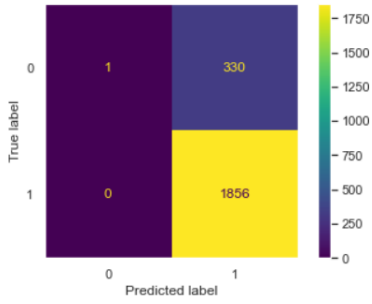

Figure 5.6: Logistic Regression Classifier Confusion Matrix without SMOTE

The confusion matrices for the Logistic Regression Classifier, figure 5.5, shows the results where SMOTE was used where the model was able to correctly predict 321 true negatives and 1833 true positives. In figure 5.6 when SMOTE is not used the model was only able to predict 1 true negative and 1856 true positives. Revealing that the Logistic Regression model is not able to predict the minority class when it is trained on the imbalanced training set. However, when using SMOTE on the training set, by having a perfectly balanced training set, helps the Logistic Regression model train on more cases for failed builds. To focus on the true negative class, which is the failed builds, the specificity metric when SMOTE is not used is 0.003% and when SMOTE is used the specificity metric is 96.9%.

5.1.4 Support Vector Machine Classifier Confusion Matrix

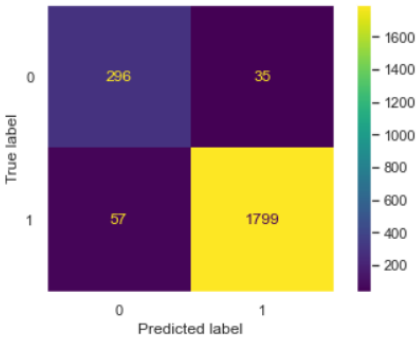

Figure 5.7: Support Vector Machine Classifier Confusion Matrix with SMOTE

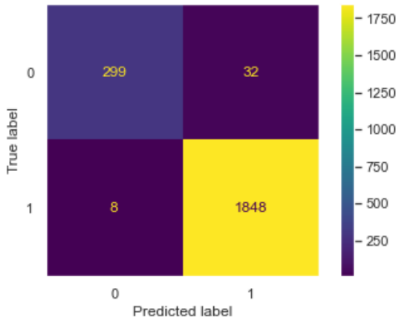

Figure 5.8: Support Vector Machine Classifier Confusion Matrix without SMOTE

The Support Vector Machine Classifier when trained on the balanced training set, figure 5.7, was able to correctly predict 296 true negatives and 1799 true positives. In figure 5.8 when SMOTE is not used on the training set the model was able to correctly predict 299 true negatives and 1848 true positives. The classifier can more accurately predict the true negative when SMOTE is not used, with 90% specificity score. With SMOTE the classifier has specificity score of 89%. When the model is not trained on the balanced training set it has a recall score of 99.8% then when SMOTE is used the recall score is 96.9%.

5.1.5 Neural Network Classifier Confusion Matrix

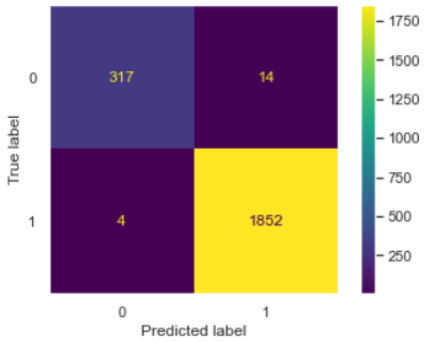

Figure 5.9: Neural Network Classifier Confusion Matrix with SMOTE

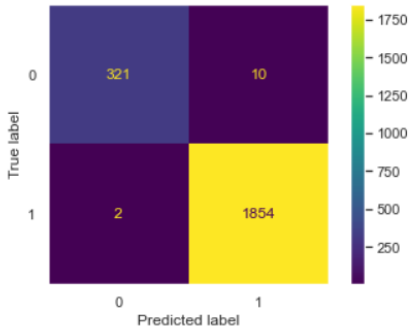

Figure 5.10: Neural Network Classifier Confusion Matrix without SMOTE

The confusion matrices for the Neural Network, figure 5.9 shows that the Neural Network classifier predicted 317 true negatives and 1852 true positives when SMOTE is used on the training set. In figure 5.10 shows that the Neural Network classifier predicted 321 true negatives and 1854 true positives without using SMOTE. The results don't differ that much between those two confusion matrices for the Neural Network. When SMOTE is not used the Neural Network classifier achieves 99.6% F1-score and with SMOTE the classifier achieves 99.5% F1-score.

5.2 Performance Metrics

In table 5.1, are the performance metrics shown for the Dummy Baseline, Random Forest, Logistic Regression, Support Vector Machine, and Neural Network without using SMOTE on the training set. What is noteworthy in these results is that the Logistic Regression model has the lowest specificity, also the recall metric is 100% which is the percentage of real positive cases that are correctly predicted positive. Neural Network achieves the best performance for all the performance metrics with 99.4% precision, 99.8% recall, 96.9% specificity, and 99.6% F1-score. The Random Forest and Support Vector Machine models are not far behind the Neural Network model.

Performance Metrics				
	precision	recall	specificity	F1-score
DC	0.8460	0.4886	0.5015	0.6195
RF	0.9819	0.9956	0.8972	0.9887
LR	0.8490	1.0	0.0030	0.9183
SVM	0.9829	0.9956	0.9033	0.9892
NN	0.9946	0.9989	0.9697	0.9967

Table 5.1: Performance metrics to evaluate the performance of the models without SMOTE

In table 5.2 are the results shown for the performance metrics when SMOTE is used on the training set. Viewing the F1-score across all the models and comparing to table 5.1, only the Dummy Baseline model achieves a higher F1-score when SMOTE is used on the training set. The other models perform slightly worse but there's not a big difference.

Performance Metrics				
	precision	recall	specificity	F1-score
DC	0.8506	0.4940	0.5135	0.6250
RF	0.9761	0.9935	0.8640	0.9847
LR	0.9945	0.9876	0.9697	0.9910
SVM	0.9809	0.9692	0.8942	0.9750
NN	0.9924	0.9978	0.9577	0.9951

Table 5.2: Performance metrics to evaluate the performance of the models with SMOTE

5.3 ROC AUC

The comparison of the models consists of using the balanced dataset, tuned hyperparameters and dimensionality reduction. The true positive rate and the false positive rate is plotted for all classification thresholds using the prediction probability to plot the ROC curve. True positives are cases of correctly predicted successful builds and false positives are cases of falsely predicted successful builds that have the true label as failed build. The ROC curve, in figure 5.11, shows a curve for all the models as well the AUC score. The horizontal axis represents the false positive rate and the vertical axis the true positive rate. The green curve represents the Support Vector Machine model, which achieves the highest AUC score of 99.6%.

The Neural Network model is represented with the red line and has the highest ROC curve in the graph and has 99% AUC score. The second highest ROC curve is the orange curve that represents the Logistic Regression model with 98,1% AUC score. The third highest ROC curve is the green curve for the Support Vector Machine model. Random Forest is represented with the blue line that has 99.2% AUC score.

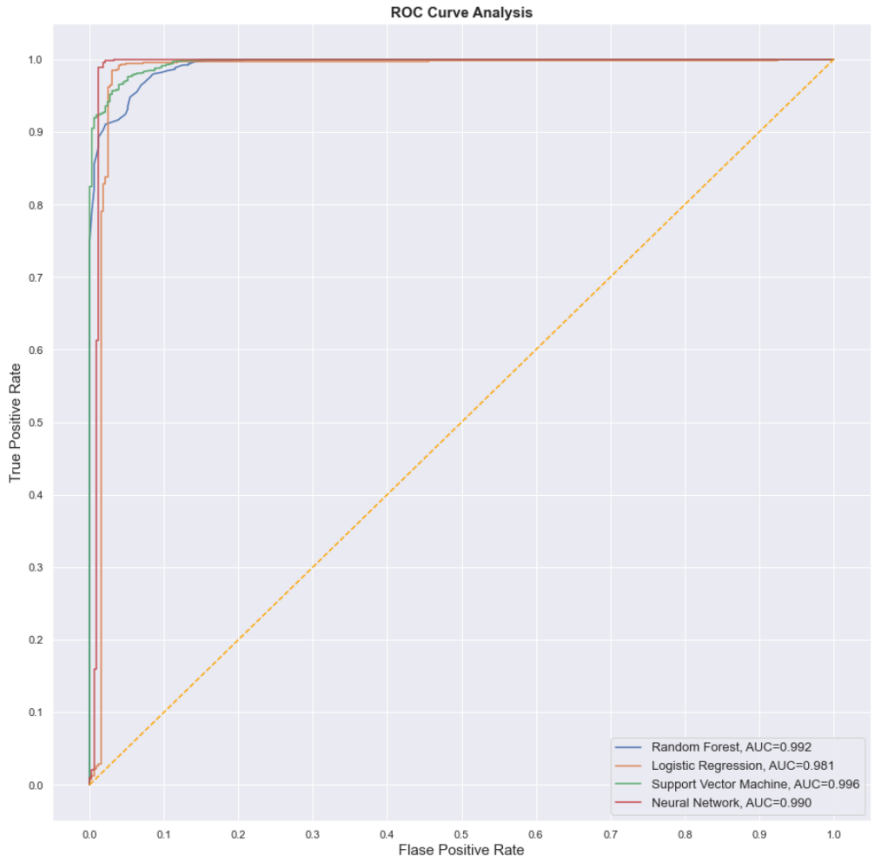

Figure 5.11: ROC curve and AUC score for all the models

5.4 Prediction

After creating proof-of-concept Machine Learning models, a short experiment was done. This experiment was done to showcase that the best performing model can predict successful or failed code builds in an efficient way. This experiment included running the prediction method from Scikit-Learn using the Neural Network model on values that the model had not seen before but the same features are used as described in chapter 3, section 3.4. The time it took to run the prediction was two milliseconds for a code change that was predicted to be successful. For code change that was predicted to fail it took three milliseconds to run the prediction method. For comparison the average code build time at the host company is 18 minutes.

5.5 Discussion

The results from the experiment indicate that the historical data available at the host company can be used to implement and train Machine Learning models with high accuracy in predicting if a code change is likely to fail or be successful during a code build without running the actual code build.

It does not seem to make a lot of difference with applying the SMOTE method to balance the training set. However, Logistic Regression is an exception there as the results show without using SMOTE the Logistic Regression classifier, where it was only able to correctly predict 1 true negative case/failed code builds. Why this is the case is an unknown factor and something that can be investigated in future work.

The other models, Random Forest, Support Vector Machine, and Neural Network achieved a better outcome without using SMOTE on the training set to balance it.

The Neural Network model achieves the highest score for both recall and specificity, or 99.8% recall and 96.9% specificity, without using SMOTE on the training set. These results show that the Neural Network model is good at predicting both the successful code builds and the failed code builds.

The results from the small prediction experiment, from section 5.4, gives an insight into the time it takes to run predictions on local hardware. If a solution would be implemented to be used in the production environment then additional time will be added, for example for an API call. A possible solution of taking this further is discussed in section 6.2.

Chapter 6

Conclusions and Future work

This chapter introduces the conclusions and future work for this research. The conclusions section 6.1, restates the topic and the thesis, summarizes the main points, and states the results. The future work section 6.2, discusses future work for carrying out potential research to continue this work as well suggestions how the results from this work can be used in production environment within the Continuous Integration pipeline.

6.1 Conclusions

With time-consuming processes in software development such as code builds that live in the Continuous Integration pipeline can have many sides effects. These side effects can decrease productivity, high resource consumption, expensive hardware resources. By improving the time of code builds effectively with a solution that can predict with good accuracy if a code build is likely to fail or be successful can be a key factor in addressing the side effects.

With this research, an experiment has been carried out by implementing four Machine Learning models, Random Forest, Logistic Regression, Support Vector Machine, and Neural Network. The raw data used to create a dataset to train these models was historical data from Jenkins, stored in ElasticSearch, and Perforce.

After carrying out data cleaning and data analysis on the raw data it revealed potential in being a good fit for creating data points to serve as features for the Machine Learning models. It was noted that the data was imbalanced, where a majority of the data included successful builds. To address the imbalanced dataset the SMOTE method was used to create synthetic samples and balancing the dataset completely, having 50% successful builds and 50%

failed builds.

The results shown in the previous chapter show that the difference between having an imbalanced dataset and a balanced dataset does not seem to have a high impact on the performance metrics. The results can differentiate between the data being used to train the models as well if a feature engineering method is used on the training data to decrease the dimensionality of the features. In this research, the PCA feature engineering method was used for dimensionality reduction to avoid overfitting the models. After comparing the results from all the models the Neural Network is the winner of outperforming Random Forest, Logistic Regression, and Support Vector Machine.

6.2 Future work

This section focuses on some of the remaining issues that should be addressed in future work. First of all, one could answer the following question: What are the expected quantitative effects of acting upon Machine Learning classification predictions, in terms of time and resource consumption?

To approach that area this work could be continued and applied into the production environment of the Continuous Integration pipeline. Before applying this work to the production environment, a proper solution must be implemented that engineers can easily use. A theoretical example of a possible solution could be a script that is triggered via Jenkins jobs before build jobs are triggered to run the scripts for the code builds. This could give faster feedback of prediction if the code change is likely to be successful or if it will fail.

There is one aspect that needs to be considered if the prediction of a code change is likely to fail, the engineer would need analytical information(error logs) on why it is likely to fail to be able to make a change to the code accordingly. Then, depending on if there is a queue of code changes waiting in Jenkins for code builds to run, the predictions can be used to shuffle the queue based on the accurate predictions made. The queue would be in the order that the code changes that are most likely to run successfully will be first in the queue and the code changes that are likely to fail to be last in the queue.

To implement a stable and more reliable solution, a bigger dataset needs to be created as the results from this work is a proof-of-concept that the historical data available at the company can be used to implement successful Machine Learning models. Another question that can be answered for future work is if the sample that was used for the experiment is a typical and representative one. Another suggestion is also to add more features, such as the authors' seniority that created the code change and continue retraining the Machine

Learning models. An important factor is also to focus on how well the Machine Learning models correctly predict that a code change is likely to fail. The reason being that if a code change is wrongfully classified as failing the code builds a developer might spend valuable time on changing the code when in fact there are no improvements needed. This again addresses the important need for error logs when it comes to future work.

As this experiment was with focus on if the available data at the host company could be used to implement a Machine Learning model that can predict with high accuracy if a code change will be successful or fail, there is another interesting aspect left to research. What identifies a successful build and a failed build. Due to time limitation there was not enough time to investigate this.

References

[1] "What is CI/CD?" Feb 2021, [Accessed: 2. Feb. 2021]. [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-ci-cd

[2] C. Gehman, "What is a monorepo?" [Accessed: 20. Mar. 2021]. [Online]. Available: https://www.perforce.com/blog/vcs/what-monorepo

[3] "Solve the hardest devops challenges with the power of perforce," [Accessed: 29. Mar. 2021]. [Online]. Available: https://www.perforce.com/

[4] A. Håkansson, "Portal of research methods and methodologies for research projects and degree projects," in *The 2013 World Congress in Computer Science, Computer Engineering, and Applied Computing WORLDCOMP 2013; Las Vegas, Nevada, USA, 22-25 July*. CSREA Press USA, 2013, pp. 67–73.

[5] 2021, [Accessed: 26. Mar. 2021]. [Online]. Available: https://www.jenkins.io/

[6] T. Ghaleb, D. Costa, and Y. Zou, "An Empirical Study of the Long Duration of Continuous Integration Builds," vol. 24, no. 2, Aug 2019. doi: 10.1007/s10664-019-09695-9

[7] M. Ananstasov, "Continuous integration (ci) explained," 2021, [Accessed: 26. Mar. 2021]. [Online]. Available: https://semaphoreci.com/continuous-integration

[8] M. WALKER, *Python Data Cleaning COOKBOOK*. PACKT PUBLISHING LIMITED, 2020. [Online]. Available: https://learning.oreilly.com/library/view/python-data-cleaning/9781800565661/B16433_Preface_Final_NM_ePUB.xhtml

[9] "7.12 Overall Launch Animation," Mar 2021, [Accessed: 31. Mar. 2021]. [Online]. Available: https://www.elastic.co

[10] G. Bonaccorso, *Machine Learning Algorithms - Second Edition*. Packt Publishing. ISBN 978-1-78934799-9. [Online]. Available: https://learning.oreilly.com/library/view/machine-learning-algorithms/9781789347999/46ad22e9-6630-4611-8463-fe1657873b1f.xhtml

[11] A. Géron, *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019. [Online]. Available: https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch07.html#ensembles_chapter

[12] D. K. Philipp Kats, *Learn Python by Building Data Science Applications*. Packt Publishing. ISBN 978-1-78953536-5. [Online]. Available: https://learning.oreilly.com/library/view/learn-python-by/9781789535365

[13] J. Brownlee, "Logistic regression for machine learning," Apr 2016, [Accessed: 26. Mar. 2021]. [Online]. Available: https://machinelearningmastery.com/logistic-regression-for-machine-learning/

[14] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition*. Sebastopol, CA, USA: O'Reilly Media, Inc. ISBN 978-1-49203264-9. [Online]. Available: https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch01.html

[15] C. C. Aggarwal, *Data Classification*. Chapman and Hall/CRC. ISBN 978-1-49876058-4. [Online]. Available: https://learning.oreilly.com/library/view/data-classification/9781466586741/K20307_C007.xhtml

[16] D. A. Pisner and D. M. Schnyer, "Chapter 6 - support vector machine," in *Machine Learning*, A. Mechelli and S. Vieira, Eds. Academic Press, 2020, pp. 101–121. ISBN 978-0-12-815739-8. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128157398000067

[17] C. C. Aggarwal, *Data Classification*. Chapman and Hall/CRC. ISBN 978-1-49876058-4. [Online]. Available: https://learning.oreilly.com/library/view/data-classification/9781466586741/K20307_C008.xhtml

[18] J. Brownlee, "4 types of classification tasks in machine learning," Apr 2020, [Accessed: 26. Mar. 2021]. [Online]. Available: https://machinelearningmastery.com/types-of-classification-in-machine-learning/

[19] O. Media, *Hands-On Artificial Intelligence for Cybersecurity*. Packt Publishing. ISBN 978-1-78980402-7. [Online]. Available: https://learning.oreilly.com/library/view/hands-on-artificial-intelligence/9781789804027/e5f4aeae-a499-4626-a817-d96c093ce2c6.xhtml

[20] E. Tsukerman, *Machine Learning for Cybersecurity Cookbook*. Packt Publishing. ISBN 978-1-78961467-1. [Online]. Available: https://learning.oreilly.com/library/view/machine-learning-for/9781789614671/5e15154e-b1ac-4bad-b4d4-1fc312ba9389.xhtml

[21] A. Parisi, *Hands-On Artificial Intelligence for Cybersecurity*. Packt Publishing. ISBN 978-1-78980402-7. [Online]. Available: https://learning.oreilly.com/library/view/hands-on-artificial-intelligence/9781789804027/e12f0379-791e-4f7d-a48c-bd22586b247c.xhtml

[22] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic Minority Over-sampling Technique," 2002. doi: 10.1613/jair.953

[23] R. Layton, *Learning Data Mining with Python - Second Edition*. Packt Publishing. ISBN 978-1-78712678-7. [Online]. Available: https://learning.oreilly.com/library/view/learning-data-mining/9781787126787/f959ac14-78e2-4015-8f32-fe4105db7387.xhtml

[24] J. Avila, *scikit-learn Cookbook - Second Edition*. Packt Publishing. ISBN 978-1-78728638-2. [Online]. Available: https://learning.oreilly.com/library/view/scikit-learn-cookbook-/9781787286382/a7ea4013-e6c0-408c-8e0d-34db7ff967e7.xhtml

[25] S. Ananthanarayanan, M. S. Ardekani, D. Haenikel, B. Varadarajan, S. Soriano, D. Patel, and A.-R. Adl-Tabatabai, "Keeping master green at scale," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019. doi: 10.1145/3302424.3303970. ISBN 9781450362818. [Online]. Available: https://doi.org/10.1145/3302424.3303970

[26] F. Hassan and X. Wang, "Change-aware build prediction model for stall avoidance in continuous integration," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017. doi: 10.1109/ESEM.2017.23 pp. 157–162.

[27] I. Saidani, A. Ouni, M. Chouchen, and M. W. Mkaouer, "Predicting continuous integration build failures using evolutionary search," *Information and Software Technology*, vol. 128, p. 106392, 2020. doi: https://doi.org/10.1016/j.infsof.2020.106392. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920301579

[28] S. Patel, "The research paradigm – methodology, epistemology and ontology – explained in simple language," Jul 2015, [Accessed: 26. Mar. 2021]. [Online]. Available: http://salmapatel.co.uk/academia/the-research-paradigm-methodology-epistemology-and-ontology-explained-in-simple-language/

[29] "Python Release Python 3.8.5," May 2021, [Online; accessed 7. May 2021]. [Online]. Available: https://www.python.org/downloads/release/python-385

[30] "Jupyter Notebooks — Anaconda 6.1.4 documentation," Apr 2021, [Online; accessed 7. May 2021]. [Online]. Available: https://team-docs.anaconda.com/en/latest/user/notebook.html

[31] "json — JSON encoder and decoder — Python 3.9.5 documentation," May 2021, [Online; accessed 7. May 2021]. [Online]. Available: https://docs.python.org/3/library/json.html

[32] "NumPy," May 2021, [Online; accessed 7. May 2021]. [Online]. Available: https://numpy.org/install

[33] "Installation — Matplotlib 3.4.1 documentation," Apr 2021, [Online; accessed 7. May 2021]. [Online]. Available: https://matplotlib.org/stable/users/installing.html

[34] "Installing and getting started — seaborn 0.11.1 documentation," Apr 2021, [Online; accessed 7. May 2021]. [Online]. Available: https://seaborn.pydata.org/installing.html

[35] "pandas - Python Data Analysis Library," May 2021, [Online; accessed 7. May 2021]. [Online]. Available: https://pandas.pydata.org/getting_started.html

[36] "Installing scikit-learn — scikit-learn 0.24.2 documentation," May 2021, [Online; accessed 7. May 2021]. [Online]. Available: https://scikit-learn.org/stable/install.html

[37] "Getting Started — Version 0.8.0," Apr 2021, [Online; accessed 7. May 2021]. [Online]. Available: https://imbalanced-learn.org/stable/install.html#getting-started

[38] A. Zheng, *Evaluating Machine Learning Models*. Sebastopol, CA, USA: O'Reilly Media, Inc. ISBN 978-1-49193244-5. [Online]. Available: https://learning.oreilly.com/library/view/evaluating-machine-learning/9781492048756

[39] H. Saleh, *Machine Learning Fundamentals*. Packt Publishing. ISBN 978-1-78980355-6. [Online]. Available: https://learning.oreilly.com/library/view/machine-learning-fundamentals/9781789803556/C11868_3_comm_Final.xhtml

[40] C. Albon, *Machine Learning with Python Cookbook*. Sebastopol, CA, USA: O'Reilly Media, Inc. ISBN 978-1-49198938-8. [Online]. Available: https://learning.oreilly.com/library/view/machine-learning-with/9781491989371

[41] S. Molin, *Hands-On Data Analysis with Pandas*. Packt Publishing. ISBN 978-1-78961532-6. [Online]. Available: https://learning.oreilly.com/library/view/hands-on-data-analysis/9781789615326/7f33060a-99bf-4bac-a73c-5a146ed60c81.xhtml

[42] G. Hackeling, *Mastering Machine Learning with scikit-learn - Second Edition*. Packt Publishing. ISBN 978-1-78829987-9. [Online]. Available: https://learning.oreilly.com/library/view/mastering-machine-learning/9781788299879/#publisher_resources

[43] V. M. Sebastian Raschka, *Python Machine Learning - Third Edition*. Packt Publishing. ISBN 978-1-78995575-0. [Online]. Available: https://learning.oreilly.com/library/view/python-machine-learning/9781789955750/Text/Chapter_6.xhtml

[44] Jul 2018, [Online; accessed 26. May 2021]. [Online]. Available: https://www.ijcai.org/Proceedings/95-2/Papers/016.pdf

[45] S. Molin, *Hands-On Data Analysis with Pandas*. Packt Publishing. ISBN 978-1-78961532-6. [Online]. Available: https://learning.oreilly.c

om/library/view/hands-on-data-analysis/9781789615326/278b45fc-d056-4d83-9929-955cbec96c14.xhtml

[46] O. Media, *Z-score standardization*. Packt Publishing. ISBN 978-1-78728760-0. [Online]. Available: https://learning.oreilly.com/library/view/feature-engineering-made/9781787287600/23a14672-9d83-4d43-a85c-b3117d7e1c9e.xhtml

[47] V. M. Sebastian Raschka, *Python Machine Learning - Third Edition*. Packt Publishing. ISBN 978-1-78995575-0. [Online]. Available: https://learning.oreilly.com/library/view/python-machine-learning/9781789955750

[48] S. Molin, *Hands-On Data Analysis with Pandas*. Packt Publishing. ISBN 978-1-78961532-6. [Online]. Available: https://learning.oreilly.com/library/view/hands-on-data-analysis/9781789615326/8f98acaa-fd00-4bf3-8fe3-e07273782a4b.xhtml

[49] Y. H. Hwang, *Hands-On Data Science for Marketing*. Packt Publishing. ISBN 978-1-78934634-3. [Online]. Available: https://learning.oreilly.com/library/view/hands-on-data-science/9781789346343/103457d5-5653-45ce-886a-918ff9040e0c.xhtml

[50] J. VanderPlas, *Python Data Science Handbook*. Sebastopol, CA, USA: O'Reilly Media, Inc. ISBN 978-1-49191205-8. [Online]. Available: https://learning.oreilly.com/library/view/python-data-science/9781491912126/ch05.html

[51] S. Molin, *Hands-On Data Analysis with Pandas*. Packt Publishing. ISBN 978-1-78961532-6. [Online]. Available: https://learning.oreilly.com/library/view/hands-on-data-analysis/9781789615326/f51ec02e-e48d-4b9e-83fb-fb716fbf1bdf.xhtml

[52] "sklearn.neural_network.MLPClassifier — scikit-learn 0.24.2 documentation," Jun 2021, [Online; accessed 10. Jun. 2021]. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

[53] "SMOTE — Version 0.8.0," May 2021, [Online; accessed 8. Jun. 2021]. [Online]. Available: https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.SMOTE.html#imblearn.over_sampling.SMOTE.fit_resample

For DIVA


```
{
"Author1": {
            "Last name": "Sigurðardóttir",
            "First name": "Sigrún Arna",
            "Local User Id": "u1cvdeok",
            "E-mail": "sasig@kth.se",
           },
"Degree": {"Educational program": "Master's Programme, Software Engineering of Distributed Systems, 120 credits"},
"Title": {
            "Main title": "Experimental Research on a Continuous Integrating pipeline with a Machine Learning approach",
            "Subtitle": "Master Thesis done in collaboration with Electronic Arts",
            "Language": "eng" },
"Alternative title": {
            "Main title": "",
            "Language": "swe"
},
"Supervisor1": {
                "Last name": "Reynisson",
                "First name": "Víðir Orri",
                "E-mail": "vidir.reynisson@dice.se",
                "Other organisation": "DICE"}
            },
"Supervisor2": {
                "Last name": "Kilander",
                "First name": "Fredrik",
                "Local User Id": "u1kv5y5m",
                "E-mail": "fki@kth.se",
                "organisation": {"L1": "School of Architecture and the Built Environment ",
                                 "L2": "Public Buildings" }
            },
"Examiner1": {
                "Last name": "Västberg",
                "First name": "Anders",
                "Local User Id": "u1ft3a12",
                "E-mail": "vastberg@kth.se",
                "organisation": {"L1": "School of Electrical Engineering and Computer Science ",
                                 "L2": "Computer Science" }
            },
"Cooperation": { "Partner_name": "Electronic Arts"},
"Other information": {
"Year": "2021", "Number of pages": "1,51"}
}
```

TRITA -EECS-EX-2021:534