Prediction of Loan Defaulters using Machine Learning Methods

Author: Nicolas Obregon

Supervisor: Dr. Daniel Buchan

Goldsmiths, University of London

This thesis is submitted in fulfillment of the requirements for a BSc Computer Science Degree

May 5th, 2022

Github Repository

https://github.com/nic-royo/Loan_Prediction

Declaration of Authorship

I, Nicolas Obregon Royo, declare that this thesis titled "*Prediction of Loan Defaulters using Machine Learning Methods*" and work presented are my own. I confirm that:

- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quotes from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Signed: Nicolas Obregon Royo

Date: May 5th, 2022

Abstract

This is a research project aimed at exploring how Artificial Intelligence models, particularly those related to Machine Learning, can predict whether a user will default on a loan or not. This dissertation consists of a Python application that implements Decision Tree, Random Forest, AdaBoost, Support Vector Machines, and Naive Bayes algorithms to a dataset consisting of information about bank loaners. This application relies heavily on techniques such as Data Visualization, Data Cleaning and Preprocessing, Machine Learning, Artificial Intelligence and Data Mining.

Acknowledgements

I would like to thank my supervisor Dr. Daniel Buchan, for supervising my project. Moreover, for acting as a ‘guide’ for me into the world of machine learning. Jumping straight into a machine learning project with little background knowledge can be complicated and overwhelming, but with the informative discussions I had with Dr. Buchan it was a smooth transition and one where I learned a massive amount.

I would also like to thank my parents and twin brother for giving me their unconditional support during the last three years, as well as Sofie, because where I am and who I am today would have not been possible without her.

Keywords

Machine Learning, Model Prediction, Loan Defaulting, Supervised Learning, Classification Task

List of Contents

List of Figures

Chapter 1

Introduction

1.1 Context and Motivation

The main motivation of this project is my interest in financial technology and machine learning. In the field of computer science, data science and AI are the subjects in which I am the most interested in as well. I therefore chose a project where I would use Python to develop a machine learning model which is also related to the financial world. In particular, digital banking and online banking were used as a main motivator for this dissertation. In the past 5 years, the use of online banks has been exponentially increasing, in Latin America, some examples are RapiCredit, EasyCredit, or Nubank, while in Europe and North America a more popular alternative would be Monzo or Revolut.

1.2 Current state of the Field

To give an example of the above mentioned banks, Nubank, a Brazilian fintech bank that has been operating for less than a decade, has quickly garnered a vast amount of users (close to 35 million only in Brazil [1]), and offers easier to use bank applications than typical normal banks, as well as reduced bureaucracy. Some of these advantages include having a credit card that is accessed through a mobile app, giving users near absolute control of the actions that they can do with their money immediately. This could be things such as canceling or freezing a card or

account, creating an account in a matter of minutes, approving payments, seeing payments, sending and receiving money, etc. [3].

Moreover, these digital banking companies are beginning to offer immediate loans. The main purpose of this is for customers to request a loan that they can immediately receive or be denied. Typically, these loans are not applicable for long term purchases such as buying a house or car, but more for daily appliances or basic needs of which the customer might not be able to afford currently. As such, these loans are generally a low amount of cash and must be paid back in a short period of time. Every bank does this differently, but as an example, Nubank offers personal loans which must be paid in 24 months, and in which the limit differs depending on the client, when they began offering this service the minimum amount was around 30 Brazilian reais (equivalent to around 5 pounds), so it is clear that they offer quick loans of not a lot of money [2]. Another example can be RapiCredit, where users can apply for a loan ranging from 110,000 to 750,000 Colombian Pesos (around 22 - 150 pounds) and pay it back in as little as 5 - 150 days. Again, it is clear that these are relatively small loans. This loan can also be approved or denied within seconds and it only takes a few minutes to make the application [4].

1.3 Gaps to Fill and Problems in the Industry

The main problem in the digital banking industry that I am trying to solve is about these loans, and particularly, about their interest rate. Loans that are offered by digital banking companies that offer low amounts of money that must be paid back in just a few months usually have very

high interest rates, RapiCredit for example has an interest rate of a whopping 25% [4]. Although banks have not officially disclosed information on why these interest rates are so high (especially when compared to typical house-buying loans, which are often around 1-5%), it can be attributed to either one of two most likely scenarios. The first scenario is that these banks want to increment their monetary gains while at the same time potentially driving away customers by having high interest rates. The second scenario is that there is a great amount of loan defaulters, in other words users who do not pay back their loan, which causes the bank to increase the interest rate as to recover the money that was lost on the unpaid loan. This project aims to fix the second scenario.

1.4 Objectives

This research project aims to create models that will make predictions on users and determine what their chances are to pay back a loan or not. Ideally, a system like the one I have developed could be used by banking companies to help make calculations and avoid having loans unpaid. Instead of creating just one model, I will create several and use hyperparameter tuning on the models to find a predictive model that has both a good precision and recall score. The dataset I have consists of 250,000 values each containing information about one user, and each having a number of variables that are used to make the prediction. This project, if added and implemented using MLOps to a bank could significantly reduce the money that is lost by the banks. Therefore the main objective is to have at least one model that can perform with precision and recall scores higher than 0.80 when tested on the testing set.

1.5 System Components

This project consists of a Python [15] application. Libraries used are Pandas [5], Matplotlib [6], Numpy [7], Scipy [8], Seaborn [9], and scikit-learn [10].

Data mining and visualization are used in the first part of the project, particularly in exploring, processing and making the data available for training. In the second part of the project, machine learning is used to create models and fit the data into them.

1.6 Basic Results

This will be explored thoroughly in the *Results* section at the end of this research paper. However, to summarize the results quickly , both decision trees and random forest models were able to reach measures of accuracy, precision and recall, all above 0.80, with ROC AUC scores also above 0.80. This is ideal in a predictive case such as the ones a banking institution would theoretically have, and is a sufficient score.

Chapter 2

Background (Literature Review)

2.1 Machine Learning

Machine learning is the science of programming computers so that they are capable of learning and identifying patterns from data. It is a subset of artificial intelligence, and to typically build machine learning models we need to do the following. Identify the problem we want to solve, look at the patterns in the variables that cause our problem, then write a detection algorithm for said patterns. Lastly, we test the model and redo it as many times as needed until we consider our results good enough [13]. In our particular case, we will train the machine with user specific data which includes an output variable. Here we hope that the machine will see that the variables have a pattern with the output variable, which is what we want to predict. Once the model has learnt these methods to predict, we can apply the models on the testing set, which is a dataset that includes the same input variables as the training dataset, but not the output variable. Therefore we can say we are seeing how the models work when being tested against unseen data[13].

The performance of machine learning classification tasks can be measured with specific measures that they will give. These are Accuracy, Precision, Recall, F1 Score and the ROC AUC Curve.

Accuracy is defined as the percentage of correct predictions in both

Precision is the ratio of true positives out of those which were predicted as positives.

Recall, or sensitivity is the detection rate of the positive class

The f1 score is the harmonic mean of precision and recall [29].

We can use a confusion matrix to better see what these measures signify.

	Actually Positive	Actually Negative
Predicted Positive	True Positives (TP)	False Positives (FP)
Predicted Negative	False Negatives (FN)	True Negatives (TN)

The function for precision is: $\frac{TP}{TP+FP}$

The function for recall is $\frac{TP}{TP+FN}$

The function for accuracy is $\frac{TP+TN}{TP+TN+FP+FN}$

The function for f1 score is $\frac{2a}{2a+b+c}$

Which metric to choose for this project as the most important is difficult. It must be kept in mind that the ultimate deciders if this project would be implemented in real life would likely be stakeholders of executives, and we do not have the resources to know what they would prefer. For consistency, I will do hyperparameter tuning searching for the best accuracy metric.

The ROC AUC Curve is another metric used to calculate how the model has distinguished between classes. The larger the area under the curve is, the better our model is performing. On the y axis lies the true positive rate and the x axis lies the false positive rate. The nearer our AUC score is to 1, the better it is.

2.2 Supervised Learning

For this task we will use Supervised Learning. This type of learning is the most commonly used method. Supervised learning occurs when our model knows what the output variable is, and we want to predict an attribute with variables. There are different ways to do supervised learning, including regression and classification. The main objective of supervised learning is that we create a model that is so well learnt that when we introduce new input data the model will predict the correct output variables [14].

2.3 Binary Classification Task

This research project will focus on doing a binary classification task. In this case the machine will be learning how to predict loan defaulting, which means we have two output variables (variables to be predicted), 0 and 1. Therefore, our output variables are a category, even if they are represented by numbers, the numbers indicate something (0 standing for users who have not defaulted and 1 for users who have, or the other way around). In contrast to a classification task, there are regression tasks, where we attempt to predict a number that can be something such as height, weight, money, etc [14].

There are different types of classification, but since we only have two output variables we will be doing binary classification.For a binary classification task to be attempted, we must have input variables, which are the variables that we will use to predict the outcome. The outcome that we want to predict is called the output variable [12]. This type of classification task attempts to

classify the items in a set into one of two groups, in our case the groups being 1: has defaulted or 0: has not defaulted.

2.4 Data Sources

The dataset is acquired from Kaggle, and is named '*Loan Prediction Based on Customer Behavior'*. The particular csv file that was used for the project is titled *Training Data.csv*. This particular file contains 12 variables. The dataset has around 250,000 rows of users.

The following are the input variables, followed by a simple explanation of each:

- ID: States the ID of the individual
- Income: States the income of the individuals
- Age: States the age of the individual
- Experience: States the experience of the user working in number of years
- Married/Single: Indicates if the user is married or single. Changed to relationship_status
- House_Ownership: indicates whether owns a house or rents
- Car_Ownership: indicates whether yes the user has a car or not
- Profession: indicates the profession of the user
- CITY: indicates what city the user lives in
- STATE: indicates what state the user lives in
- CURRENT_JOB_YRS: indicates the years the person has been at at their current job
- CURRENT_HOUSE_YRS: Indicates the number of years the person has owned a house for

Lastly, the following is our output variable:

- Risk_Flag: indicates whether 0, has not defaulted, 1 has defaulted

For this research project, the purpose of the dataset was to provide data on which machine learning models could be built on and decisive conclusions could be reached, therefore, it is not a main concern if the data is real or not. Although there are many real datasets available, many of these include seriously raw data, and since this project was made to explore machine learning methods instead of doing data cleaning these datasets were dropped in favor of cleaner ones, particularly this one.

2.5 Scikit-learn

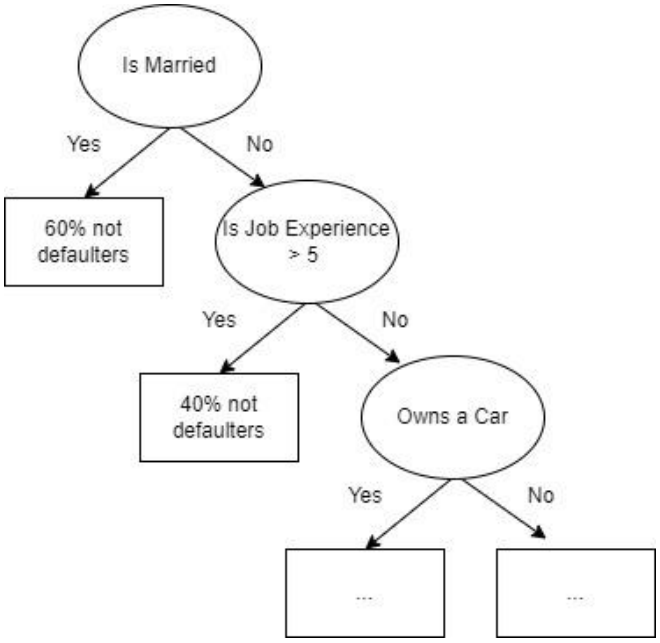
The Python library on which most if not all of the machine learning code will be implemented is scikit-learn [10]. Scikit-learn is an amazing and incredibly thorough open source library made for data analysis and machine learning. It is one of the most widely used libraries for machine learning related tasks, supporting the implementation of multiple classification and regression tasks, as well as providing sufficient tools to do data analysis.

2.5.1 Decision Trees

The first machine learning model I will be implementing is a decision tree. A decision tree can be used both for regression and classification tasks and works by having a tree-like model which makes decisions, with nodes representing each condition. The case below, a simplified example of our case, would have the model go through the training set and see the percentage of people

who fit a specific variable. Decision trees are drawn with the root at the top. Depending on the conditions of the nodes, the tree makes branches into different nodes when it reaches an edge, and can continue growing until all conditions have reached an edge, resulting in enormous trees which often lead to overfitting. To avoid this I will be using hyperparameter tuning on the '*Methods*' section to see what is the ideal maximum depth for the tree to have that will yield the best results [16]. The ending of a tree can also

FIGURE 1.1: Decision Tree Example



2.5.2 Random Forest Classifier

Random Forest Algorithms works by creating a number of decision trees, making an ensemble out of them, and reaching a consensus based on the average of all the trees. The big difference between decision trees and random forest algorithms is that the trees in the random forest

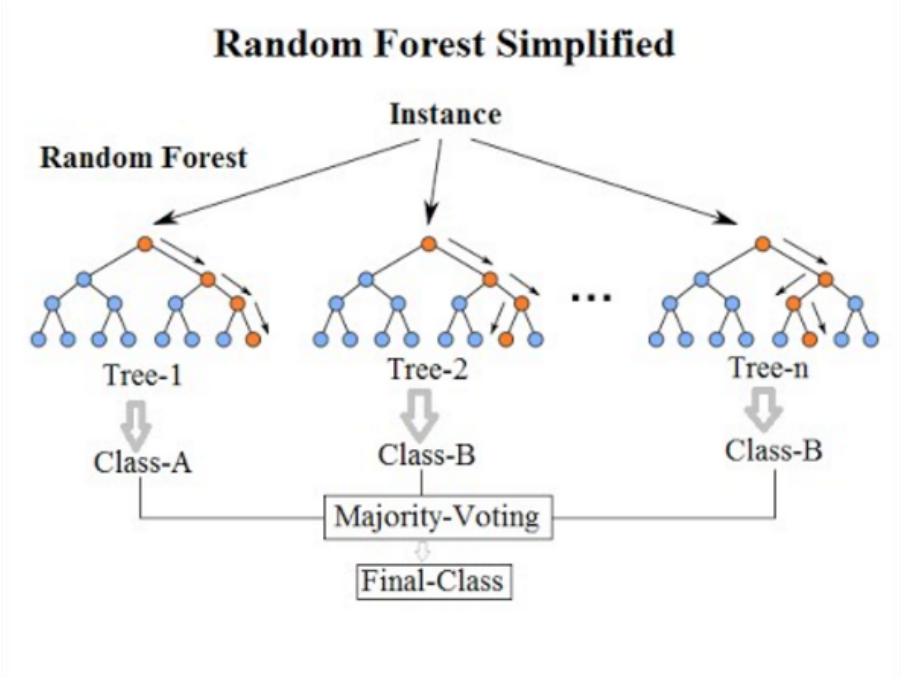
algorithm do not evaluate all the input variables in the nodes, but instead only a random sample size *M* (which is a hyperparameter and will be further talked about in the hyperparameter tuning section in the '*Methods*' chapter) of attributes competed on each node. Random forests operate under the thought of 'multiple opinions are better than one'. It finds the average of multiple estimates instead of only one (like decision trees do). Random forests use a method called 'bagging', where we train an amount of M different trees on different parts of the dataset (these are chosen randomly), which are then made into an ensemble from which we can calculate an average. This average, although more computationally expensive than a single decision tree, can often give us better and more precise scores. Random forests include more randomness so the trees are less correlated, this is because we want the decision tree classifiers to be more independent from each other, and less correlated. The more independent, the better judgment we get [18].

The equation that computes the ensemble is the following [18]:

$$f(x) = \sum_{m=1}^{M} \frac{1}{M} f_m(x)$$

Where f_m will be the tree number m [18].

FIGURE 1.2: Random Forest Example

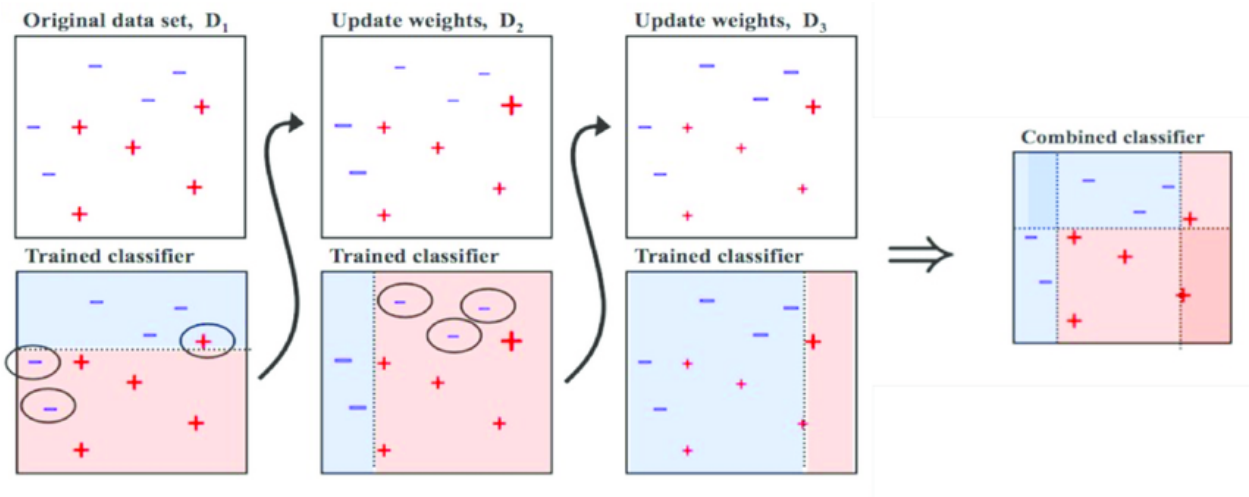


2.5.3 AdaBoost Algorithm

Boosting algorithms are another method which can often be better than random forests, as it sometimes provides better results in prediction. Boosting algorithms have models which are dependent on each other, with the idea that models complement each other, boosting also uses voting (in our case since it is classification, otherwise for regression it would use averaging). Boosting however is iterative, meaning each model is influenced by the previous one. In our case, we will be using the AdaBoost algorithm since it is designed specifically for classification. Once again, we build multiple classifiers except now the models are not independent unlike those in random forests. On the other hand, they are now dependent on each other. This works by having one model built, and then seeing which of its data points have errors occur when evaluated on the training set. These data points are referred to as misclassification.

We want the next model to do a better prediction on the misclassified data points, so for the points that were previously misclassified in the previous model, their weight is increased, and they now have a higher importance. Doing this iteratively will reduce the error, and we will continue training models until a lower error is achieved/. As mentioned, when we begin all instances have the same weight, but as the learning algorithm develops, and sees the errors in the data of each model, for the next model these errors have higher weights, and the others have lower weights. As each model is built it focuses on making correct classifications of the error points. This leads us typically to a classification model that in some cases can be excellent as it has removed the errors. The Adaboost algorithm has the disadvantage of having low enough weights removed entirely from the dataset, causing information loss [19].

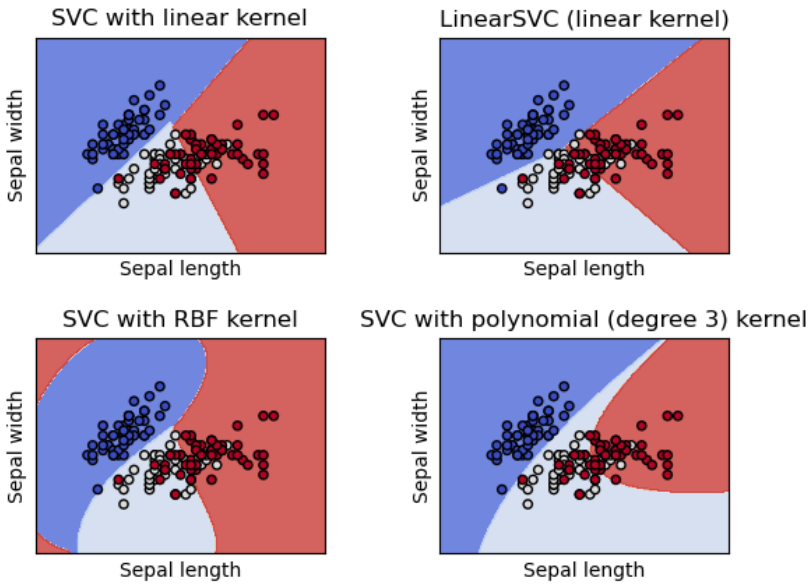
FIGURE 1.3: AdaBoost Example



2.5.4 Support Vector Machines

Support vector machines work by finding a linear hyperplane (with a decision boundary) that separates the data, like figure 1.4 below, and distinctly classifies the data. We have multiple ways of dividing the data, and the ideal way to do so is by finding the hyperplane that maximizes the margin, so it must be as dense as it can, which helps with noise and outliers. Linear support vector machines are represented by a linear model (where the hyperplane will be a R^2 line, with R being the number of input features), while there are other types such as nonlinear support vector machines where the decision boundary is not linear, meaning that we often need to transform the data into a higher dimensional space, called the kernel trick (wen we have R^3, the hyperplane is a plane). Support vector machines are robust to noise but have difficulty handling missing values. Overfitting is dealt with by maximizing the margin of the decision boundary, and they often have a high computational expense when building the model [22].

FIGURE 1.4: Support Vector Machines Example



Our specific ase has over 12 input features, meaning our support vector machine will be of high dimensionality. To maximize the margin between the hyperplane and the data points, a function called the hinge loss function is used.

2.5.5 Naive Bayes

The Naive Bayes classifier is a type of probabilistic classifier. This classifier uses Bayes theorem (along the 'naive') assumption of conditional independence among predictors to make supervised learning algorithms [23]. Bayesian statistics rely on using the acquisition of new information to update the probability of a classification occurring. Naive Bayes specifically focuses on factors that describe a variable being independent from each other (for example, a car has wheels, doors and a steering wheel, but these three objects can be found in other things, but when combined together they contribute independently to the probability that this object is a car).

The Naive Bayes equation is as follows:

$$P(c|x) = \frac{P(x|c)P(c)}{P(x)}$$

Where $P(c|x) = P(x_1|c) * P(x_2|c) *...* P(x_n|c) * P(c)$, and $P(c|x)$ is the probability that class c will occur if the predictor x occurs. In our case, c is whether a user defaults or not, and x are the input variables that each user has, with P being the probability [24]. So in other words:

- $P(c|x)$: Probability of c being true if x is true
- $P(x|c)$: Probability of x being true if $c$is true
- $P(c)$: Probability of c being true
- $P(x)$: Probability of x being true

Naive Bayes is incredibly fast and works well with multi class prediction, and is ideal to use when we believe our variables are independent. In our case, even though the variables might not necessarily be independent (i.e. having a car might be more relevant for people living in big cities, or having a higher salary might be correlated with certain jobs or levels of education) our variables should be considered independent as otherwise it might lead to artificial intelligence bias. This is something that is ideal to avoid as it can cause serious repercussions, including racism and sexism towards the users of a product which was developed with this model. It is also ideal to use when we have multiple categorical variables [24].

2.6 Google Colaboratory

The code was mainly run and written in Google Colaboratory. This was due to the service offering free access to GPUS which could potentially run many of the machine learning models much quicker than one computer. At the beginning of the project decision trees were taking 5 minutes to run and random forests more than 20 minutes, without even having apple grid search to the algorithms. Due to this it seemed like a sensical action to take [25].

2.7 Hyperparameter Tuning and GridSearchCV

Machine learning models have parameters that control how the learning process of each model is made. Different models have specific parameters that can be altered to receive different results. These results can vary wildly, so it is ideal to do a search of different parameters on a model as it can greatly increase the performance. One way of exploring a model's different parameters is by doing Grid Search. Scikit-learn's GridSearchCV function [31] which does an exhaustive search over parameters that are specified by the user. This function explores specific combinations of hyperparameter values (which are provided by the user) to find the best scores. This world by creating values from a grid of parameters specified with the param_grid parameter, and then it fits on a dataset all of the mentioned combinations of parameters and keeps the best one [32].

Chapter 3

Methods

3.1 Data Overview, Cleaning and Preprocessing

Before beginning to train models, significant data preparation needed to be done. In the following sections what was done will be explained in great detail

3.1.1 Libraries

Several libraries were imported to create the project. Some of them were not used anymore in the final build but were initially imported to fix bugs, for contextual reasons I will include them in this section.

First, Pandas [5], which is used thoroughly all throughout the project. Pandas is used for data manipulation and analysis. In our project it is most mostly used to use the read_csv function to load the training dataset into Colaboratory.
Second, matplotlib [6] is a library used for data visualization and was often used in this project to visualize results, particularly using the matplotlib.pyplot extension.
Numpy [7] was also used for debugging but is no longer present in the final build. It can be used to work with arrays and matrices.

Scipy [8] was also imported to use in the beginning of the project but as advancements were made it was rendered inefficient by other libraries more appropriate for this project. It is used for statistical programming with Python.

Seaborn [9] is a highly advanced library used for visualization, it was used multiple times throughout this project.

Most importantly, Scikit-learn [10] is an amazing library used to create machine learning models. It brings with it numerous models that are easy to implement and tune.

3.1.2 Data Type Analysis

Data cleaning is a primordial step when doing machine learning. To start, we saw what the data types where, and found out that most where either int64 data types or object types

FIGURE 3.1: Data Types Pre-Encoding

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 252000 entries, 1 to 252000
Data columns (total 12 columns):
 #   Column             Non-Null Count   Dtype
---  ------             --------------   -----
 0   Income             252000 non-null  int64
 1   Age                252000 non-null  int64
 2   Experience         252000 non-null  int64
 3   Married/Single     252000 non-null  object
 4   House_Ownership    252000 non-null  object
 5   Car_Ownership      252000 non-null  object
 6   Profession         252000 non-null  object
 7   CITY               252000 non-null  object
 8   STATE              252000 non-null  object
 9   CURRENT_JOB_YRS    252000 non-null  int64
 10  CURRENT_HOUSE_YRS  252000 non-null  int64
 11  Risk_Flag          252000 non-null  int64
dtypes: int64(6), object(6)
memory usage: 25.0+ MB
```

This presents problems, particularly with the objects that are classified as objects. A classification task will work better with numerical data. Besides this, the rename() function was used to change the name of the **Married/Single** variable to **Relationship_Status**.

3.1.3 Encoding

To fix the variable data types a method from sklearn preprocessing was brought in, this being the label encoder. With this, we can transform the object variables into int64 variables, this will mean that when previously we had under the CITY variable different objects each denoting a different city, now we will have numbers for each city. This process was done for the following variables: Relationship_Status, House_Ownership, Car_Ownership, Profession, CITY, STATE. Figure 3.2 displays what they look like afterwards.

FIGURE 3.2: Example of Data Types After Encoding

Id	Income	Age	Experience	Relationship_Status	House_Ownership	Car_Ownership	Profession	CITY	STATE	CURRENT_JOB_YRS	CURRENT_HOUSE_YRS	Risk_Flag
1	1303834	23	3	1	2	0	33	251	13	3	13	0
2	7574516	40	10	1	2	0	43	227	14	9	13	0
3	3991815	66	4	0	2	0	47	8	12	4	10	0
4	6256451	41	2	1	2	1	43	54	17	2	12	1
5	5768871	47	11	1	2	0	11	296	22	3	14	1
...	...	...	...	...	...	...	...	...	...	...	...	...
251996	8154883	43	13	1	2	0	45	162	28	6	11	0
251997	2843572	26	10	1	2	0	3	251	13	6	11	0
251998	4522448	46	7	1	2	0	17	144	14	7	12	0
251999	6507128	45	0	1	2	0	27	233	18	0	10	0
252000	9070230	70	17	1	2	0	44	26	22	7	11	0

252000 rows × 12 columns

Furthermore, by using the info() function once again we can see that the data types have been correctly transformed into int64 data types.

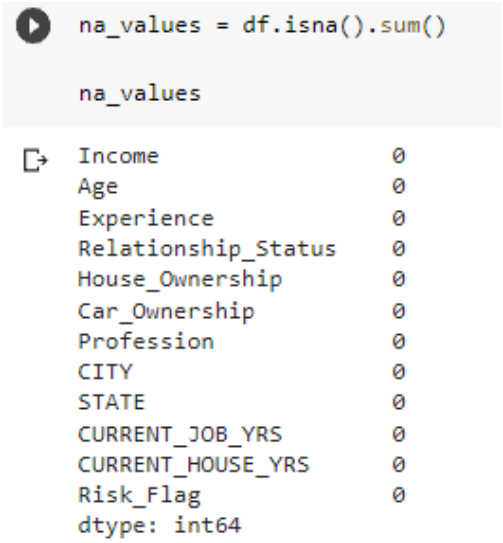
FIGURE 3.3: Data Types Post-Encoding

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 252000 entries, 1 to 252000
Data columns (total 12 columns):
 #   Column               Non-Null Count   Dtype
---  ------               --------------   -----
 0   Income               252000 non-null  int64
 1   Age                  252000 non-null  int64
 2   Experience           252000 non-null  int64
 3   Relationship_Status  252000 non-null  int64
 4   House_Ownership      252000 non-null  int64
 5   Car_Ownership        252000 non-null  int64
 6   Profession           252000 non-null  int64
 7   CITY                 252000 non-null  int64
 8   STATE                252000 non-null  int64
 9   CURRENT_JOB_YRS      252000 non-null  int64
 10  CURRENT_HOUSE_YRS    252000 non-null  int64
 11  Risk_Flag            252000 non-null  int64
dtypes: int64(12)
memory usage: 25.0 MB
```

3.1.4 Missing Values

The next natural step in data preprocessing is finding missing values and dealing with them as these can heavily alter the way the models work. Luckily the dataset that was chosen has no missing or NA values, as can be seen when using the isna().sum() function on the data frame. Having said that, often when running the code iteratively to see if there are different results the data frame will be affected and a couple of values will disappear. We are unsure of why this occurs but as a failsafe have implemented code that drops any NA values should there be any.

FIGURE 3.4: Missing Values



```
na_values = df.isna().sum()

na_values
```

```
Income                0
Age                   0
Experience            0
Relationship_Status    0
House_Ownership       0
Car_Ownership         0
Profession            0
CITY                  0
STATE                 0
CURRENT_JOB_YRS       0
CURRENT_HOUSE_YRS     0
Risk_Flag             0
dtype: int64
```

3.1.5 Describe Function

The next step is using the describe() function to see the variables and their main information. Now that the variables have all been encoded this function will display all variables whereas otherwise it would not display the object variables. As can be seen in figure 3.5, there are no missing values as the count for all variables is the same. However, we can see that there are some irregularities in the **Income** as the lowest Income and maximum Income highly differ.

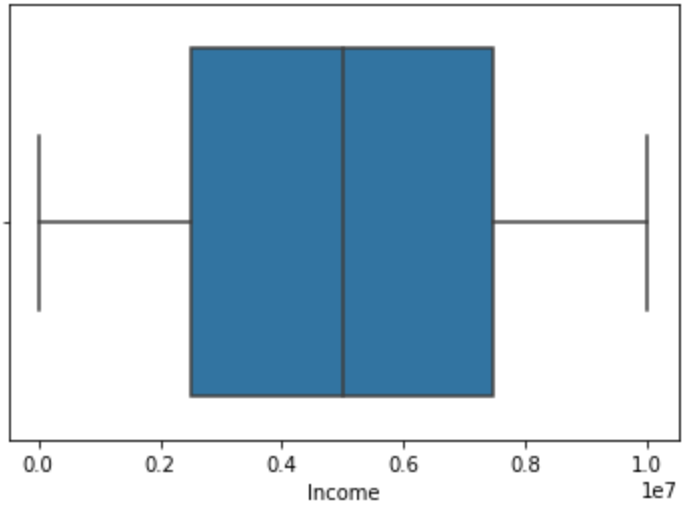
FIGURE 3.5: Variables Described

	Income	Age	Experience	Relationship_Status	House_Ownership	Car_Ownership	Profession	CITY	STATE	CURRENT_JOB_YRS	CURRENT_HOUSE_YRS	Risk_Flag
count	2.520000e+05	252000.000000	252000.000000	252000.000000	252000.000000	252000.000000	252000.000000	252000.000000	252000.000000	252000.000000	252000.000000	252000.000000
mean	4.997117e+06	49.954071	10.084437	0.897905	1.891722	0.301587	25.276746	158.137675	13.808952	6.333877	11.997794	0.123000
std	2.878311e+06	17.063855	6.002590	0.302774	0.391880	0.458948	14.728537	92.201736	9.372300	3.647053	1.399037	0.328438
min	1.031000e+04	21.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	10.000000	0.000000
25%	2.503015e+06	35.000000	5.000000	1.000000	2.000000	0.000000	13.000000	78.000000	6.000000	3.000000	11.000000	0.000000
50%	5.000694e+06	50.000000	10.000000	1.000000	2.000000	0.000000	26.000000	157.000000	14.000000	6.000000	12.000000	0.000000
75%	7.477502e+06	65.000000	15.000000	1.000000	2.000000	1.000000	38.000000	238.000000	22.000000	9.000000	13.000000	0.000000
max	9.999938e+06	79.000000	20.000000	1.000000	2.000000	1.000000	50.000000	316.000000	28.000000	14.000000	14.000000	1.000000

3.1.6 Irregularities

What was previously mentioned about the Income variable is further explored here where we attempt to see if this is an actual irregularity where outliers might be present or not. To analyze this a simple box plot was created, where we can clearly see that there are no actual outliers and the data is fine as is.

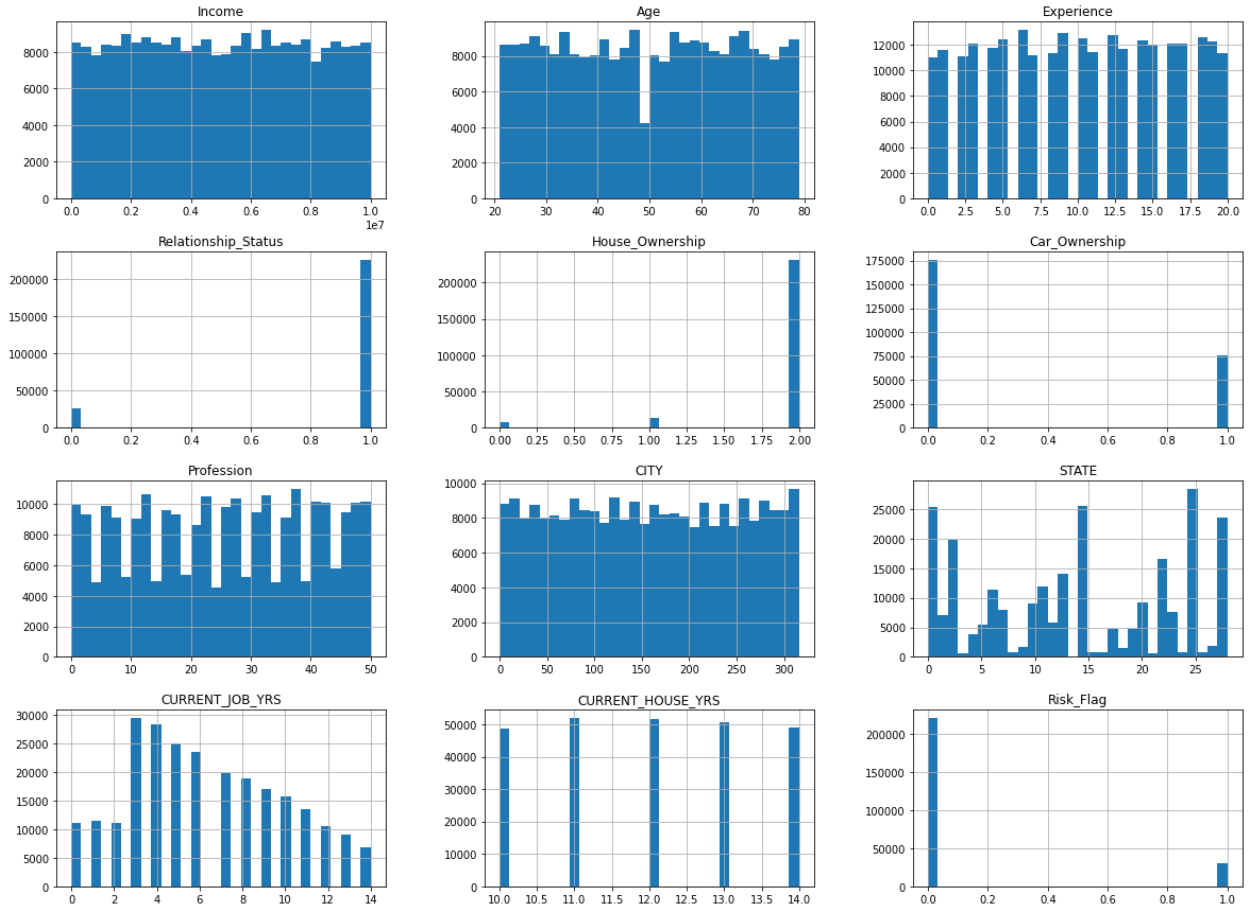
FIGURE 3.6: Income Boxplot



3.1.7 Analysis

Next, we can use the hist() function and matplotlib to display histograms of all the variables. We can see that there are no major discrepancies among most variables. However, there are significant differences in the Relationship_status, House_Ownership, Car_Ownership and Risk_Flag variables. This is a challenge because such huge differences can create problems in our models as the data is highly unbalanced. This is most important to fix in the Risk_Flag variable, our input variable.

FIGURE 3.7: Visual Analysis of the Variables

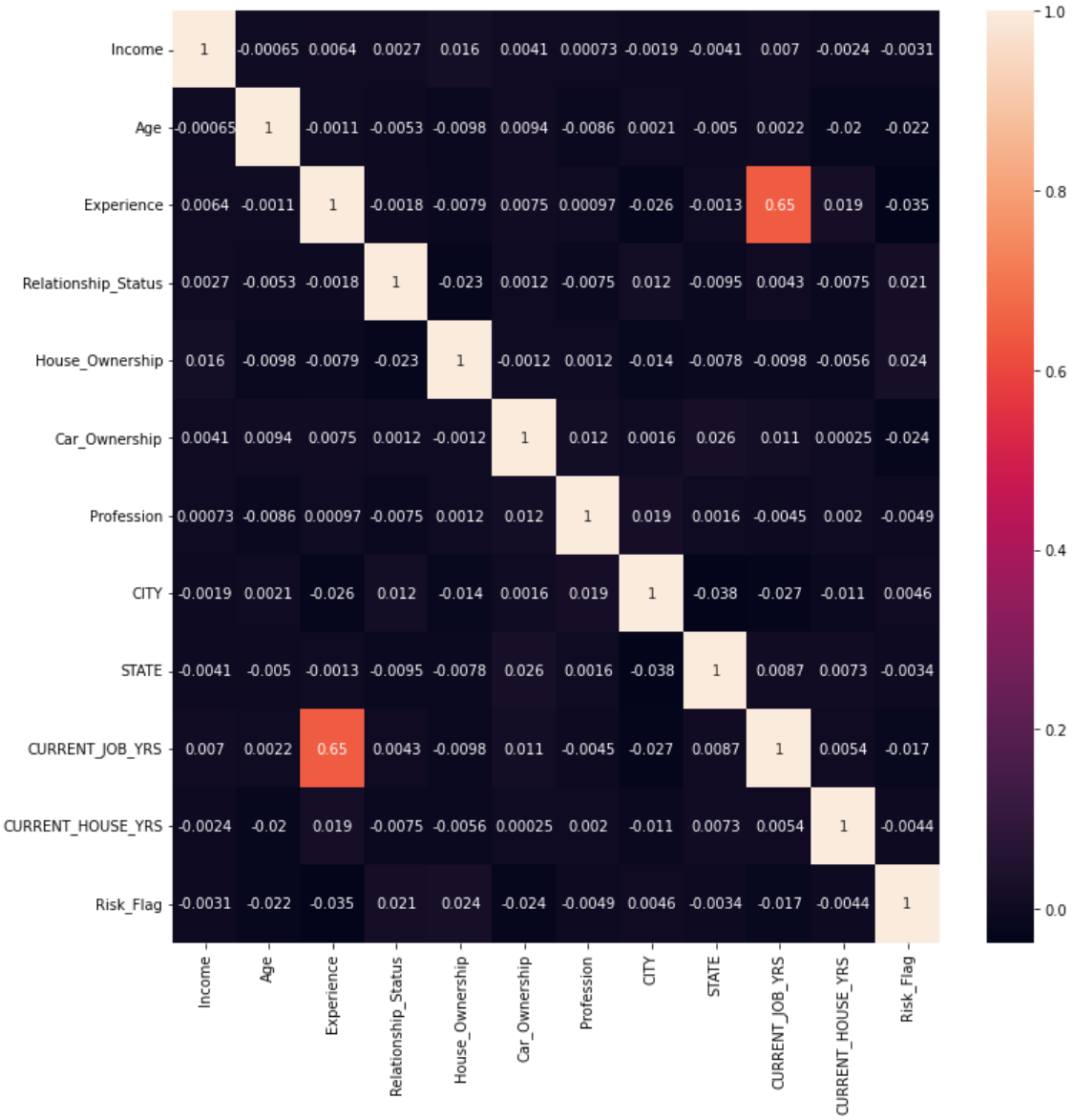


3.2 Correlation

Next, we want to see if we will have to do feature selection in the future. To start, we used the corr() function and seaborn to display a graph showing the correlations between variables. Luckily there are no significant correlations, with most of them being absolutely meaningless. There is a strong correlation between the Experience Variable and the CURRENT_JOB_YRS, which makes sense and can be ignored. Most importantly, we want none of the variables to be correlated with the RIsk_Flag variable because this could skew our models. If we had other

correlated values higher than 0.5, we would have to consider doing feature selection to have these fixed and not have a higher importance on the models than other variables. If a variable would have correlation scores higher than 0.85 we would consider dropping the variable altogether. We have used the standard pearson correlation method.

FIGURE 3.8: Correlation



3.3 Rescaling of Data

Now we do rescaling of the data. THe main purpose of this is so any values that greatly differ between the variables do not affect the model. For example, the highest value of the Income variable is over 9,999, however, the highest value of the Car_Ownership variable is 2, therefore it might be interpreted as being much less important than the Income one.

We use the MinMaxScaler [30] to scale each feature individually so its range is between 0 and 1. The results can be seen below in figure 3.9

FIGURE 3.8: Correlation

Id	Income	Age	Experience	Relationship_Status	House_Ownership	Car_Ownership	Profession	CITY	STATE	CURRENT_JOB_YRS	CURRENT_HOUSE_YRS	Risk_Flag
1	0.129487	0.034483	0.15	1.0	1.0	0.0	0.66	0.794304	0.464286	0.214286	0.75	0
2	0.757206	0.327586	0.50	1.0	1.0	0.0	0.86	0.718354	0.500000	0.642857	0.75	0
3	0.398564	0.775862	0.20	0.0	1.0	0.0	0.94	0.025316	0.428571	0.285714	0.00	0
4	0.625263	0.344828	0.10	1.0	1.0	1.0	0.86	0.170886	0.607143	0.142857	0.50	1
5	0.576454	0.448276	0.55	1.0	1.0	0.0	0.22	0.936709	0.785714	0.214286	1.00	1
...	...	...	...	...	...	...	...	...	...	...	...	...
251996	0.815303	0.379310	0.65	1.0	1.0	0.0	0.90	0.512658	1.000000	0.428571	0.25	0
251997	0.283620	0.086207	0.50	1.0	1.0	0.0	0.06	0.794304	0.464286	0.428571	0.25	0
251998	0.451682	0.431034	0.35	1.0	1.0	0.0	0.34	0.455696	0.500000	0.500000	0.50	0
251999	0.650356	0.413793	0.00	1.0	1.0	0.0	0.54	0.737342	0.642857	0.000000	0.00	0
252000	0.906933	0.844828	0.85	1.0	1.0	0.0	0.88	0.082278	0.785714	0.500000	0.25	0

252000 rows × 12 columns

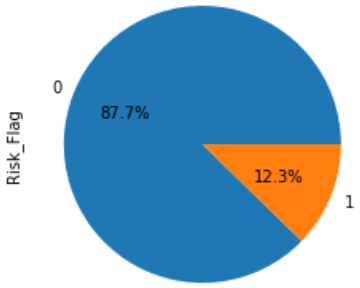
3.4 Stratification

When we split the dataset into a training and testing set, this split is done randomly. This can cause heavy imbalancement issues as one set can take fewer or more of the output variable which will seriously skew our results, most likely giving us good accuracies but poor precision and recall scores. This will be dealt with below in chapter 3.6 where we can create the sets.

3.5 Undersampling / Dealing with Imbalanced Data

The way we deal with the Imbalanced data is through undersampling, as seen in figure 3.9, our original dataset has many more instances of one output variable class than the other (0 has 88% and 1 has 12%).

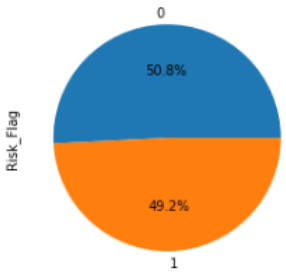
FIGURE 3.9: Imbalanced Output Variable Classes



This can be fixed by doing undersampling, this will make the class 0 have a similar amount to that of class 1.

FIGURE 3.10: Balanced Output Variable Classes

```
class0 = df[df['Risk_Flag'] == 0].sample(32000)

class1 = df[df['Risk_Flag'] == 1]

df = pd.concat([class0, class1], axis = 0)

df['Risk_Flag'].value_counts().plot(kind = 'pie', autopct = "%.1f%%")
plt.show()
```



3.6 Training and Testing Set

Now we can begin splitting the dataset into a training and testing dataset.

We will make a new variable X which is the dataframe with the Risk_Flag variable dropped and a Y variable which is the dataframes Risk_Flag variable. This will then be used in scikit-learn's train_test_split function to define what X_train, X_test, Y_train and Y_test are.

Something to note is that now that we have done undersampling, we have 93,000 values instead of the original 252,000.

FIGURE 3.11: Creation of testing and training datasets

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.2,
                                                    random_state = 42, stratify = Y)


print("X_train shape: {}".format(X_train.shape))
print("X_test shape: {}".format(X_test.shape))
print("Y_train shape: {}".format(Y_train.shape))
print("Y_test shape: {}".format(Y_test.shape))
```

```
X_train shape: (50396, 11)
X_test shape: (12600, 11)
Y_train shape: (50396,)
Y_test shape: (12600,)
```

I will be stratifying the Y variable here (which is the Risk_Flag variable, our output class) so each set takes the same amount of output variables. The split will be done 80/20 meaning that 80% of the data will be used for training and 20% of it for testing. We will set the random_state to be 42 so we can replicate the code and always receive the same results. Otherwise, we could receive different ones due to the randomness of each run.

3.7 Model Creation

Now the code is ready to create the models.

3.7.1 Decision Tree

To begin, we did a decision tree, one of the simplest but most efficient machine learning models. To build this model first I created a variable DTreeClf that uses the DecisionTreeClassifier function with the entropy criterion [33]. Then I fit this variable with the X values and Y values that were created when I split the dataset into testing and training sets. Then a new variable predsdtc was created which predicts the X_test variable on the DTreeClf variables. This can be seen below in figure 3.12.

FIGURE 3.12: Creation of Decision Tree

```
from sklearn import tree
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score


DTreeClf = tree.DecisionTreeClassifier(criterion = 'entropy')
DTreeClf = DTreeClf.fit(X.values, Y.values)
predsdtc = DTreeClf.predict(X_test)
print("accuracy_score: " + str(accuracy_score(Y_test, predsdtc)))
print("precision_score: " + str(precision_score(Y_test, predsdtc)))
print("recall_score: " + str(recall_score(Y_test, predsdtc)))
print("f1: " + str(f1_score(Y_test, predsdtc)))
```

This decision tree will produce amazing scores. It gives an accuracy of 0.95, a precision score of 0.91 and a recall score of 0.95.

3.7.1.1 Hyperparameter Tuning

However, we can further attempt to produce better results. To do this we used GridSearchCV to explore different parameters. The two parameters that were explored were criterion and max_depth.

Criterion measures the quality of the split and here we can test two different parameters which calculate the impurity of a node [33]. These are Gini and Entropy. Gini explores the frequency that mislabeling occurs in an element of our data when random labeling occurs, while entropy calculates the disorder of the target features [34].

Max_depth on the other hand is the maximum depth of the tree, meaning how long the distance between the root node and leaf node is [33]. In our case, we explored both criterions and several max depths, ranging from 4 to 150. We did 5 cross validations and set the desired scoring as 'accuracy'. And the results were that the best criterion was gini and the best max_depth was 50.

Furthermore, implementing these parameters into a decision tree and fitting it provided the following measures: an accuracy of 0.86, precision of 0.84 and recall score of 0.88. What is interesting is that these scores are less than when we implemented a decision tree without hyperparameter tuning. Efforts were done to find out why this was the case but results were inconclusive and we are still not sure of why this happened.

FIGURE 3.13: Decision Tree GridSearchCV

```
[ ]  from sklearn.model_selection import GridSearchCV

     parameter_grid = [
          {'criterion':['gini','entropy'],
           'max_depth':[4,5,6,7,8,9,10,11,12,15,20,30,40,50,70,90,120,150]}]

     grid_search = GridSearchCV(DTreeClf, parameter_grid, cv=5,
                                scoring='accuracy',
                                return_train_score=True)

     grid_search.fit(X_train, Y_train)
     print('best parameter values', grid_search.best_params_)
     print('best estimator', grid_search.best_estimator_)
```

```
best parameter values {'criterion': 'gini', 'max_depth': 50}
best estimator DecisionTreeClassifier(max_depth=50)
```

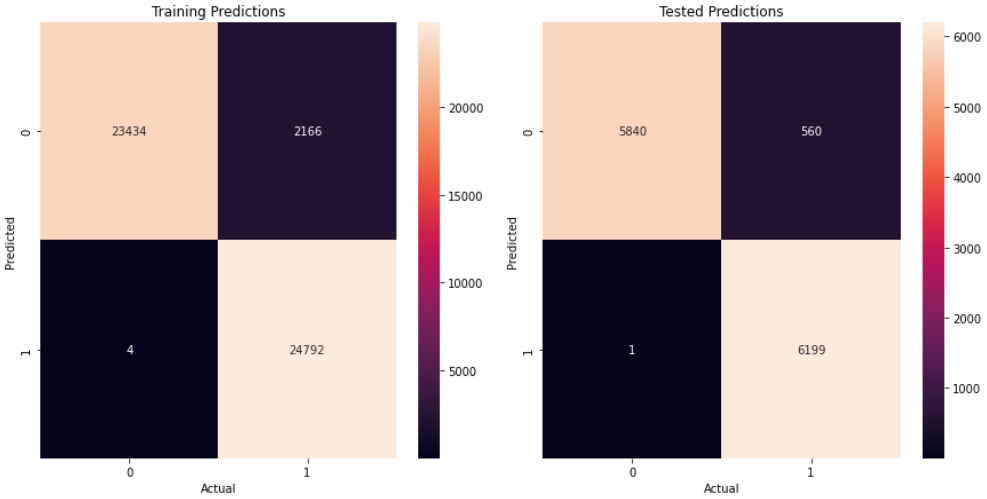
```
[ ]  best_DTreeClf=grid_search.best_estimator_
     pred_Y=best_DTreeClf.predict(X_test)

     print('\n accuracy', accuracy_score(Y_test, pred_Y))
     print('\n precision', precision_score(Y_test, pred_Y))
     print('\n recall (sensitivity)', recall_score(Y_test, pred_Y))
     print('\n f1', f1_score(Y_test, pred_Y))
```

```
 accuracy 0.8629365079365079

 precision 0.8470131885182312

 recall (sensitivity) 0.8804838709677419

 f1 0.8634242783708975
```

3.7.1.2 Interpretation

Using sklearn's classification_report module we can create confusion matrices of the training predictions and testing predictions, and the results can be seen in figure 3.14

FIGURE 3.14: Decision Tree Confusion Matrix

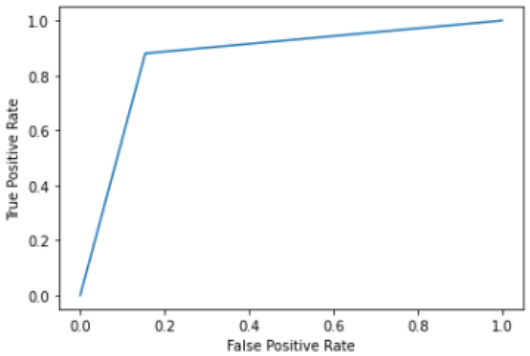


We can also create a ROC AUC curve and see that we have a score of 0.86, which is a good score as it is relatively close to 1.

FIGURE 3.15: Decision Tree ROC AUC Curve

```
import sklearn.metrics as metrics
from sklearn.metrics import roc_auc_score,roc_curve

print('\n ROC AUC Score', roc_auc_score(Y_test, pred_Y))

fpr, tpr, _ = metrics.roc_curve(Y_test, pred_Y)

plt.plot(fpr,tpr)
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```

```
 ROC AUC Score 0.8632106854838709
```



3.7.2 Random Forest

We follow similar steps when doing the next classifiers as we did with the Decision Tree (i.e. create a variable that fits the X and Y values into a machine learning function from scikit-learn, then predict using the X_test variable). This time we use the Scikit-learn Random Forest Classifier [35]. It gives us a score of 0.95 accuracy, 0.91 precision and 0.99 recall.

FIGURE 3.16: Random Forest Implementation

```
from sklearn.ensemble import RandomForestClassifier
rForestClf=RandomForestClassifier()

rForestClf = rForestClf.fit(X.values, Y.values)
predsdtc = rForestClf.predict(X_test)
print("accuracy_score: " + str(accuracy_score(Y_test, predsdtc)))
print("precision_score: " + str(precision_score(Y_test, predsdtc)))
print("recall_score: " + str(recall_score(Y_test, predsdtc)))
print("f1: " + str(f1_score(Y_test, predsdtc)))
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:4: Data
  after removing the cwd from sys.path.
/usr/local/lib/python3.7/dist-packages/sklearn/base.py:444: UserWarn
  f"X has feature names, but {self.__class__.__name__} was fitted wi
accuracy_score: 0.9554761904761905
precision_score: 0.9171475070276668
recall_score: 0.9998387096774194
f1: 0.9567096226560691
```

3.7.2.1 Hyperparameter Tuning

The hyperparameters that we explored with the Random Forest algorithm were once again criterion (gini and entropy) and max_features. Max_features is the number of features that it will consider when looking for what the best split is [35]. Once again we do 5 cross validations and seek the best accuracy metric. We explore 4 different max_features values between 2 to 6.

FIGURE 3.17: Random Forest GridSearchCV

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'max_features': [2, 3, 4, 6],
    'criterion' :['gini', 'entropy']}

grid_search = GridSearchCV(rForestClf, param_grid, cv=5,
                           scoring='accuracy',
                           return_train_score=True,
                            n_jobs=-1)

grid_search.fit(X_train, Y_train)
print('best parameter values', grid_search.best_params_)
print('best estimator', grid_search.best_estimator_)

cvres = grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.mean(mean_score), params)
```

```
best parameter values {'criterion': 'gini', 'max_features': 6}
best estimator RandomForestClassifier(max_features=6)
0.845602817109911 {'criterion': 'gini', 'max_features': 2}
0.8458607477121218 {'criterion': 'gini', 'max_features': 3}
0.8459401029486108 {'criterion': 'gini', 'max_features': 4}
0.8464560074616875 {'criterion': 'gini', 'max_features': 6}
0.8451067282751973 {'criterion': 'entropy', 'max_features': 2}
0.8454837261822081 {'criterion': 'entropy', 'max_features': 3}
0.8453250038977789 {'criterion': 'entropy', 'max_features': 4}
0.8459004085974767 {'criterion': 'entropy', 'max_features': 6}
```

```
best_rForestClf=grid_search.best_estimator_
pred_Y=best_rForestClf.predict(X_test)

print('\n accuracy', accuracy_score(Y_test, pred_Y))
print('\n precision', precision_score(Y_test, pred_Y))
print('\n recall (sensitivity)', recall_score(Y_test, pred_Y))
print('\n f1', f1_score(Y_test, pred_Y))
```

```
 accuracy 0.8488888888888889

 precision 0.8700895933838731

 recall (sensitivity) 0.8145161290322581

 f1 0.8413862045984671
```

We now get good results. An accuracy score of 0.84, precision of 0.87 and recall score of 0.81. Like with the decision tree, these are once again worse than when we fitted the model without doing grid search, this remains inconclusive and guarantees further analysis in the future.

3.7.2.2 Interpretation

Once again, we can create confusion matrices that display the results, as well as a ROC AUC curve with a score of 0.84, which is once again a good result.

FIGURE 3.18: Random Forest Confusion Matrix

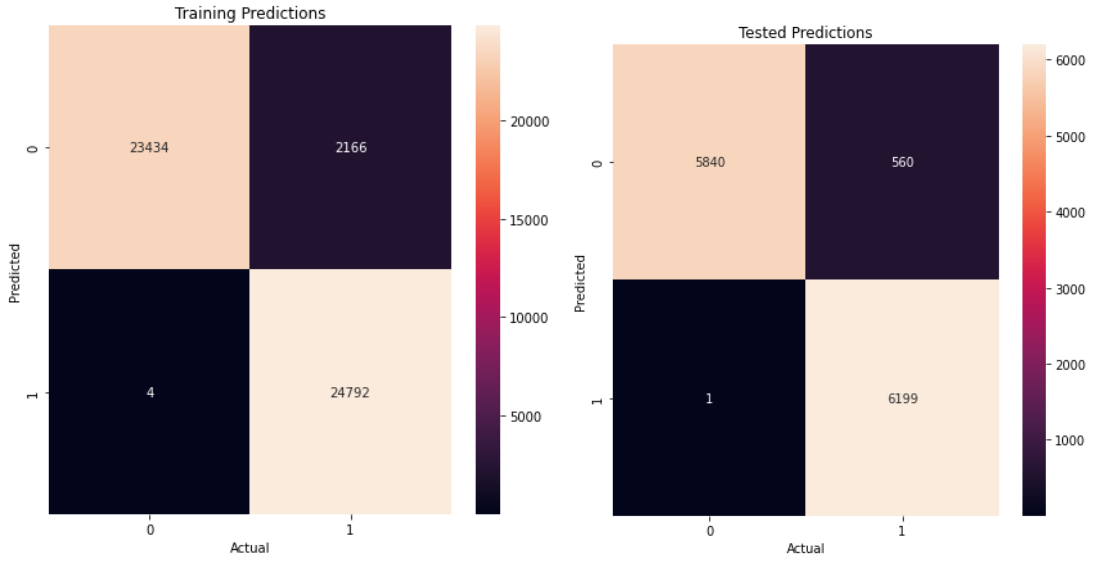
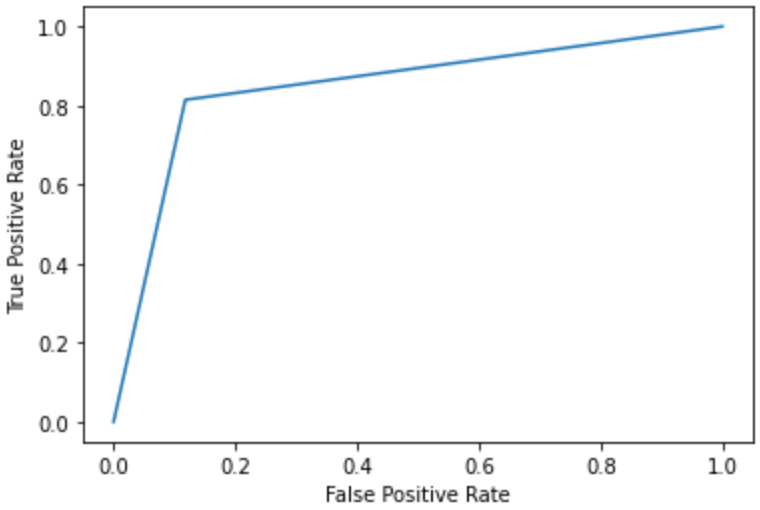


FIGURE 3.19: Random Forest ROC AUC Curve



3.7.3 AdaBoost Algorithm

Now we can move on to the AdaBoost algorithm. Once again we follow the same steps to implement the Scikit-learn AdaBoostClassifier() function [36].

Unlike the previous two classifiers, AdaBoost initially gives us an accuracy of 0.56, not the best results.

FIGURE 3.20: AdaBoost Classifier Implementation

```
[ ]  from sklearn.ensemble import AdaBoostClassifier

     ABoostClf = AdaBoostClassifier()
     ABoostClf.fit(X, Y)
     y_pred = ABoostClf.predict(X_train)
     accuracy = ABoostClf.score(X_train, Y_train)
     accuracy
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/ut
  y = column_or_1d(y, warn=True)
0.5647868878482419
```

3.7.3.1 Hyperparameter Tuning

When we do hyperparameter tuning on AdaBoost, we will do grid search on the following parameter: learning_rate. Initially, we also considered doing grid search on the n_estimators parameter but having as few as two parameters to explore heavily increased the computational time so this was dropped. The learning_rate hyperparameter is the weight each classifier gets per boosting iteration. Higher learning rates means more reach classifier contributes more[36]. The n_estimators parameter is the max number of estimators that define when boosting is stopped [36]. This parameter could be further explored with and implemented in GridSearch if higher computational power was available. Once again we do 5 cross validations and search for the best

'accuracy' score. We explore three different values for the learning rate, 0.01, 0.1 and 1. Note that the n_estimators grid parameter is present but commented out. Now the results we get are an accuracy of 0.56, precision score of 0.55 and recall score of 0.51. These scores are significantly lower than the Decision Tree Classifier and Random Forest Classifier and once again the accuracy is lower than before we did grid search.

FIGURE 3.21: AdaBoost Classifier GridSearchCV

```
from sklearn.model_selection import GridSearchCV
param_grid = {
     #'n_estimators': np.arange(10,200,10),
     'learning_rate': [0.01, 0.1, 1]
 }
grid_search = GridSearchCV(ABoostClf, param_grid, scoring='accuracy', cv=5, n_jobs=-1)
grid_search.fit(X_train, Y_train)
grid_search.best_params_
print('best parameter values', grid_search.best_params_)
print('best estimator', grid_search.best_estimator_)

cvres = grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.mean(mean_score), params)
```

```
best parameter values {'learning_rate': 1}
best estimator AdaBoostClassifier(learning_rate=1)
0.5338718607130651 {'learning_rate': 0.01}
0.5454997228246062 {'learning_rate': 0.1}
0.5566511798852556 {'learning_rate': 1}
```

```
best_ABoostGrid=grid_search.best_estimator_
pred_Y=best_ABoostGrid.predict(X_test)

cm=confusion_matrix(Y_test, pred_Y) # confusion matrix
print('confusion matrix,  classes order is 0 and 1, actual values on rows, predicted values on columns \n', cm)
print('\n accuracy', accuracy_score(Y_test, pred_Y))
print('\n precision', precision_score(Y_test, pred_Y))
print('\n recall (sensitivity)', recall_score(Y_test, pred_Y))
print('\n f1', f1_score(Y_test, pred_Y))
```

```
confusion matrix,  classes order is 0 and 1, actual values on rows, predicted values on columns
 [[3886 2514]
 [3023 3177]]

 accuracy 0.5605555555555556

 precision 0.5582498682129678

 recall (sensitivity) 0.5124193548387097

 f1 0.5343537128921033
```

3.7.3.2 Interpretation

We can again do a confusion matrix and plot a ROC AUC curve, with the ROC AUC curve having a score of 0.55.

FIGURE 3.22: AdaBoost Classifier Confusion Matrix

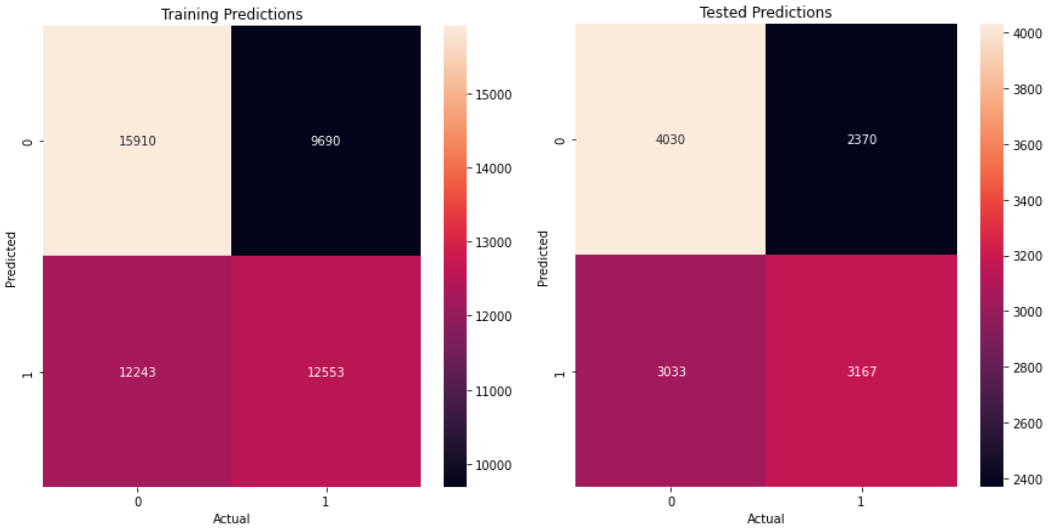
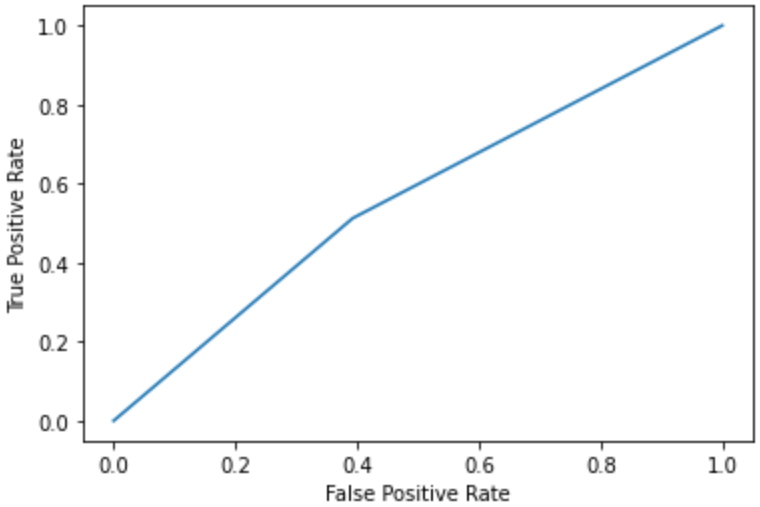


FIGURE 3.23: AdaBoost Classifier ROC AUC Curve



3.7.4 Support Vector Machines

For Support Vector Machines we do the same as when fitting previous models, and see that we receive an accuracy of 0.62 [37].

FIGURE 3.24: Support Vector Machine Implementation

```
from sklearn.svm import SVC

SVCClf = SVC()
SVCClf.fit(X, Y)
y_pred = SVCClf.predict(X_train)
accuracy = SVCClf.score(X_train, Y_train)
accuracy
```

```
/usr/local/lib/python3.7/dist-packages/skle
  y = column_or_1d(y, warn=True)
0.6279665052781966
```

3.7.4.1 Hyperparameter Tuning

When doing Grid Search with this classifier, we once again seek the best accuracy score and do 5 cross validations. This time we explore three different parameters. The first, C, is the parameter that defines the strength of the regularization, we will explore the values 0.1, 1 and 10, these can only be positive [37]. The next parameter is gamma, which is the kernel coefficient, in our case, we grid search 0.01, 0.1 and 1. The last parameter is kernel, which specifies the kernel type that is used, in our case we do only ‘linear’.

Initially The idea was to do linear as well as rbf but the computational time was too long so we cut back on this aspect. This time the accuracy was 0.53, precision score was 0.52 and recall score was 0.55. Once again, these scores are suboptimal, especially when compared to the

random forest and decision tree classifiers. It is also once again a lower accuracy than without the grid search, at this point this is getting very confusing.

FIGURE 3.25: Support Vector Machine GridSearchCV

```
param_grid = {'C': [0.1, 1, 10],
              'gamma': [1, 0.1, 0.01],
              'kernel': ['linear']
              }
grid_search = GridSearchCV(SVCClf, param_grid, scoring='accuracy', cv=5, n_jobs=-1)
grid_search.fit(X_train, Y_train)
grid_search.best_params_
print('best parameter values', grid_search.best_params_)
print('best estimator', grid_search.best_estimator_)

cvres = grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.mean(mean_score), params)
```

```
best parameter values {'C': 1, 'gamma': 1, 'kernel': 'linear'}
best estimator SVC(C=1, gamma=1, kernel='linear')
0.5345860834329432 {'C': 0.1, 'gamma': 1, 'kernel': 'linear'}
0.5345860834329432 {'C': 0.1, 'gamma': 0.1, 'kernel': 'linear'}
0.5345860834329432 {'C': 0.1, 'gamma': 0.01, 'kernel': 'linear'}
0.5348242042625165 {'C': 1, 'gamma': 1, 'kernel': 'linear'}
0.5348242042625165 {'C': 1, 'gamma': 0.1, 'kernel': 'linear'}
0.5348242042625165 {'C': 1, 'gamma': 0.01, 'kernel': 'linear'}
0.5347051428634424 {'C': 10, 'gamma': 1, 'kernel': 'linear'}
0.5347051428634424 {'C': 10, 'gamma': 0.1, 'kernel': 'linear'}
0.5347051428634424 {'C': 10, 'gamma': 0.01, 'kernel': 'linear'}
```

```
best_SVCClf = grid_search.best_estimator_
pred_Y = best_SVCClf.predict(X_test)

cm=confusion_matrix(Y_test, pred_Y) # confusion matrix
print('confusion matrix,  classes order is 0 and 1, actual values on rows, predicted values on columns \n', cm)
print('\n accuracy', accuracy_score(Y_test, pred_Y))
print('\n precision', precision_score(Y_test, pred_Y))
print('\n recall (sensitivity)', recall_score(Y_test, pred_Y))
print('\n f1', f1_score(Y_test, pred_Y))
```

```
confusion matrix,  classes order is 0 and 1, actual values on rows, predicted values on columns
 [[3267 3133]
 [2742 3458]]

 accuracy 0.5337301587301587

 precision 0.52465483234714

 recall (sensitivity) 0.557741935483871

 f1 0.5406926745367837
```

3.7.4.2 Interpretation

Once again, we do the confusion matrix and ROC AUC Curve, which has a score of 0.53. Once again a suboptimal score.

FIGURE 3.26: Support Vector Machine Confusion Matrix

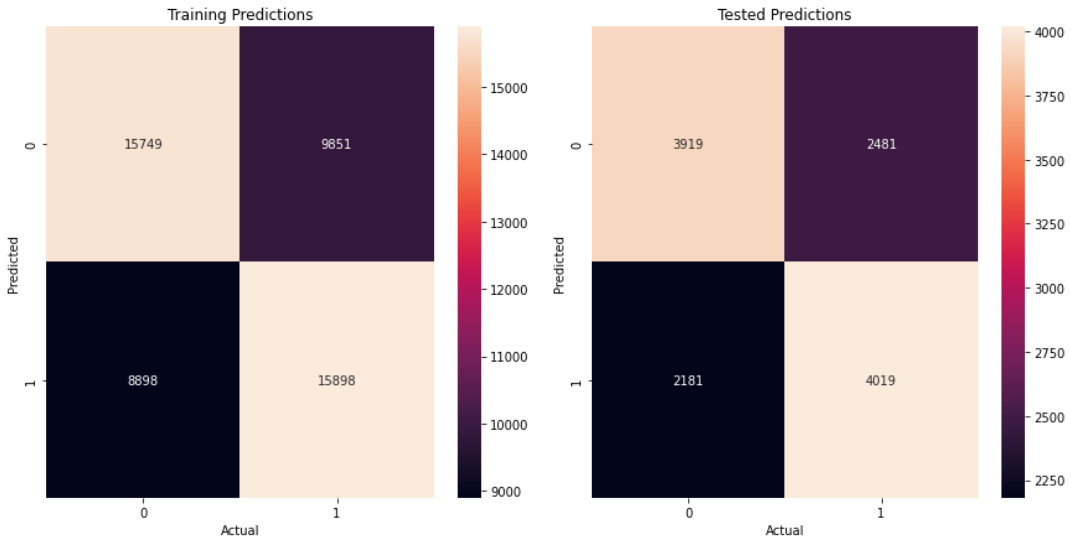
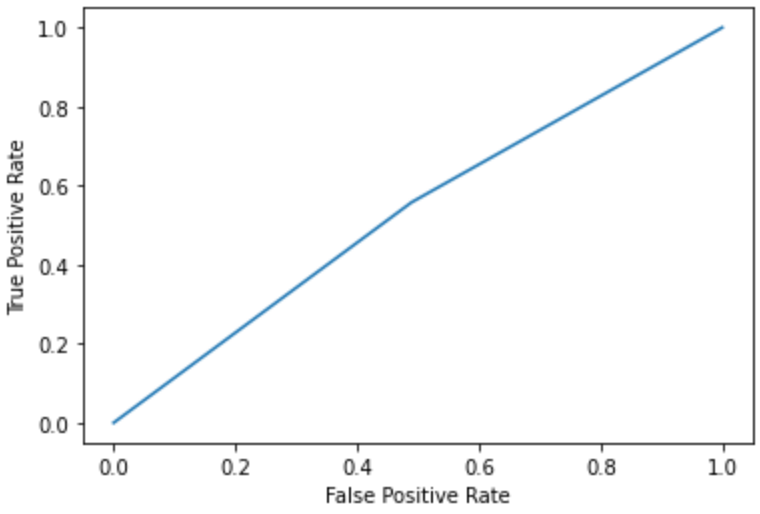


FIGURE 3.27: Support Vector Machine ROC AUC Curve



3.7.5 Naive Bayes

Lastly, we implemented a Naive Bayes Algorithm. In our case we used Scikit-learn's GaussianNB() [38]. This provided us with an accuracy of 0.53, a precision score of 0.51 and recall score of 0.74.

FIGURE 3.28: Naive Bayes Classifier Implementation

```
from sklearn.naive_bayes import GaussianNB

NaiveBayesClf = GaussianNB().fit(X_train, Y_train)
```

```
pred_Y = NaiveBayesClf.predict(X_test)
```

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score

#accuracy_score = accuracy_score(Y_test, pred_Y)
#print (accuracy_score)

print('\n accuracy', accuracy_score(Y_test, pred_Y))
print('\n precision', precision_score(Y_test, pred_Y))
print('\n recall (sensitivity)', recall_score(Y_test, pred_Y))
print('\n f1', f1_score(Y_test, pred_Y))
```

```
 accuracy 0.5355555555555556

 precision 0.5194543828264758

 recall (sensitivity) 0.7493548387096775

 f1 0.6135763338615954
```

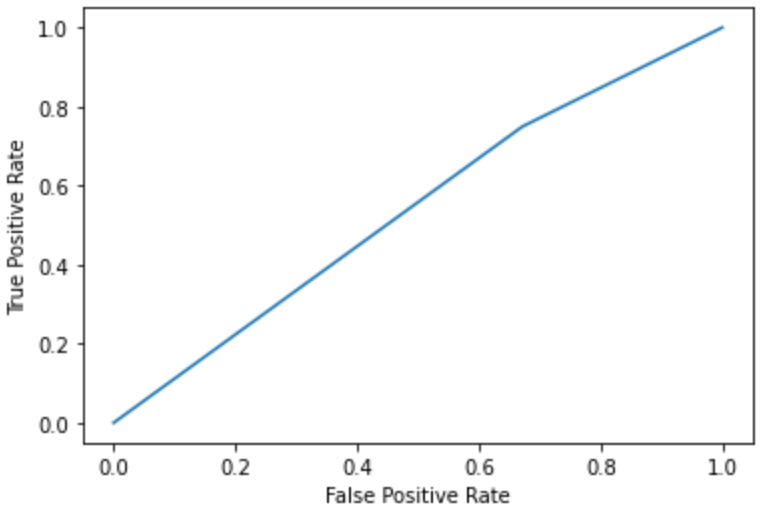
3.7.5.1 Hyperparameter Tuning

Naive Bayes does not have hyperparameters, so unfortunately grid search is not possible on this classifier.

3.7.5.2 Interpretation

We can do a ROC AUC curve and see that the score is 0.53, so not good again.

FIGURE 3.29: Naive Bayes ROC AUC Curve



3.7.6 Pre Undersampling Results

When this project was started. Due to my lack of knowledge in the subject the models were first created without implementing undersampling techniques, meaning that the scores that were received were highly skewed and extremely suboptimal. THe GItHub repository including this code can be found in the following link [https://github.com/nic-royo/Loan_Prediction/blob/main/LoanPrediction.ipynb]. Not that the project at this point was far from done so this .ipynb file is very incomplete.

It was decided that these results would be included in this report for fun, and to signify the importance of undersampling. Also to remember that every step of a machine learning project has to be as perfect as possible as even a few lines of code can deem it absolutely useless.

In this case, the Decision Tree had an accuracy score of 0.15, recall score of 0.95 and precision score of 0.12. But when tuned with GridSearch that changed to an accuracy of 0.88, with recall being 0.46 and precision 0.52.

The random forest behaved similarly, with a good accuracy of 0.79, but dismal scores for both precision (0.10) and recall (0.09). When doing hyperparameter tuning with GridSearchCV the results were also similar to the Decision Tree Classifier, with accuracy being 0.89, but precision being 0.60 and recall 0.52.

AdaBoost gave an accuracy of 0.87, and when attempting hyperparameter tuning it took more than 5 hours to complete so the process was stopped. It was also at this point that the failure was discovered and therefore I stopped working with these results.
The support vector machine model had not yet been developed in this file.

Chapter 4

Results

4.1 Total Results

The results were good in some parts, but confusing in other parts. On the one hand, the accuracy, precision and recall measures that the decision tree and random forest classifiers gave were absolutely perfect, they were all higher than 0.80 after parameter tuning. This would in many cases be considered sufficient to be implemented in real life. However, the Naive Bayes, Support Vector Machine and AdaBoost algorithms were less than satisfactory, all given median scores of 0.50-0.60.

In terms of computational power, it is hard to calculate which is more expensive than the other because this project was made in Google Colaboratory. This means that when the code ran it would run quicker or slower depending on the internet signal. Much of this code was run in student housing blocks where the internet signal can oftentimes be abysmal.

Having said that, I am more than satisfied with the scores that the Decision Tree and Random Forest classifiers gave us. When this project was started, the main objective was to get scores higher than 0.80, and this was achieved in two cases.

The github repository for this project can be found at

https://github.com/nic-royo/Loan_Prediction. Particularly, the finished file is

FinishedPrediction.ipynb

Chapter 5

Discussion

5.1 Results Discussion

The results are more than satisfactory. By having all scores be higher than 0.80 on two different models is extremely well. Whether it would be satisfactory enough for a bank enough to be further developed and implemented into a pipeline to actually determine what customers receive loans or not, is hard to tell and it would be necessary to get in contact with a bank or expert in the field.

Overall however, the results indicate that a system like this would be ready to be deployed in mobile bank applications.

Within the broader literature, this project shines due to how thorough it is. When initial investigation was done over loan prediction using machine learning, the vast majority of the results are blog posts or github projects. These blog posts also all missed several important aspects. Of academic papers, the results were similar, one particular report missed doing any hyperparameter tuning [26], others making the huge mistake of only measuring the accuracy metric [27][28], or exploring only a few classifier models. Therefore I am satisfied with how this specific project delves deeper into the models, how they work, what the results are and more importantly what the results and what metrics are important mean.

Chapter 6

Limitations and Future Work

6.1 Limitations

Limitations found were mostly due to lack of knowledge. When first starting this project I had not had a single course in AI or machine learning so getting up to date on all the necessary subjects was the most difficult. I believe that if I did this project now this year after having had multiple courses covering the theory my final product would have been much better. Other limitations were the dataset, ideally for this sort of work a larger dataset could be better but that would require more time to clean the raw data and more computational power. Another limitation was hyperparameter tuning. While it would have been interesting to explore as many parameters as possible when doing hyperparameter tuning, this was seriously increasing the time it took for each model to run through. The last and most important limitation was the fact that all classifiers had better accuracy before we did grid search on them, after grid search for some reason they had lower accuracy. Why this occurred is still unclear and unfortunately it was unsolvable for now.

6.2 Future Work

In terms of future work, there are multiple things to consider.

First, it would be interesting to create the same machine learning model but with a more detailed dataset. Particularly, one with 100+ variables and millions of user rows. A dataset such as this

would require much more time to develop and would be ideal to clean with a team of developers. Another thing to mention would be the computational expense. While the code made in this dissertation can certainly run on a computer or google colaboratory, running a code with 100+ variables would definitely need much more computational power.

Besides this, I would like to have more time to explore other classifiers and their theories. What would be the most interesting but I don't think is feasible as I imagine it would take another year would be to use MLOps to actually develop a model that can be directly applied to a product. I hope that in the future after graduation I can someday work in MLOps and do this very thing with a team of people.

Chapter 7

Conclusion

7.1 Conclusion and Self Reflection

This project was an immense challenge to develop, but also an extremely rewarding one. When I first began I believed I would finish with average models with horrible performance. However, I was very surprised to see how well some of the models worked, and that when tested on the testing set they performed extremely well.

Doing this project was a joy, and I learned that I love to code machine learning classifiers. I hope to further develop my skills in my Masters and further make a much more advanced project for my Masters thesis.

Working on this project was extremely challenging, but also fun and incredibly rewarding. I have learned that this subject is very complicated and requires a lot of attention to detail to do well. So I would not be surprised if there is an obvious, silly little mistake that I have committed somewhere in my code. Overall I am happy with how this dissertation turned out however.

Bibliography

1. *Nubank Ultrapassa OS 35 Milhões de Clientes no Brasil*. Nubank. (2021, March 14). Retrieved April 15, 2022, from https://blog.nubank.com.br/nubank-ultrapassa-35-milhoes-clientes/
2. Higa, P. (2019, February 8). Nubank Começa a oferecer Empréstimo Pessoal. Tecnoblog. Retrieved April 15, 2022, from https://tecnoblog.net/noticias/2019/02/08/nubank-emprestimo-pessoal-juros/
3. *Nubank Digital Account*. Nubank. (n.d.). Retrieved April 15, 2022, from https://nubank.com.br/en/digital-account/
4. *RapiCredit Official Website* https://www.rapicredit.com/
5. *Pandas 1.4.2:* https://pandas.pydata.org/
6. *Matplotlib 3.5.2:*https://matplotlib.org/
7. *Numpy 1.22.3:*https://numpy.org/
8. *Scipy 1.8.0:*https://scipy.org/
9. *Seaborn 0.11.2:*https://seaborn.pydata.org/
10. *Scikit-learn 1.0.2:*https://scikit-learn.org/stable/
11. *Loan Prediction Based on Customer Behavior* by Subham Surana: https://www.kaggle.com/datasets/subhamjain/loan-prediction-based-on-customer-behavior?select=Training+Data.csv
12. Brownlee, J. (2019, August 8). *Machine learning terminology from statistics and Computer Science*. Machine Learning Mastery. Retrieved April 15, 2022, from

https://machinelearningmastery.com/data-terminology-in-machine-learning/#:~:text=Those%20columns%20that%20are%20the,also%20called%20the%20response%20variable.

13. Murphy, K. P. (2021). Chapter 1.2 Supervised Learning. In *Machine learning: A probabilistic perspective*. essay, MIT Press.
14. Brownlee, J. (2020, August 20). *Supervised and unsupervised machine learning algorithms*. Machine Learning Mastery. Retrieved April 15, 2022, from https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/
15. *Python 3.9*: https://www.python.org/
16. Witten, I. H., Frank, E., & Hall, M. A. (2011). Chapter 3.3 Trees. In Data Mining: Practical Machine Learning Tools and Techniques. essay, Elsevier.
17. *Random Forest Image*, Wikimedia Image, https://community.tibco.com/wiki/random-forest-template-tibco-spotfirer-wiki-page
18. Murphy, K. P. (2021). Chapter 16.2.5 Random Forests. In *Machine learning: A probabilistic perspective*. essay, MIT Press.
19. Witten, I. H., Frank, E., & Hall, M. A. (2011). Chapter 8.4 Boosting. In Data Mining: Practical Machine Learning Tools and Techniques. essay, Elsevier.
20. Alto, V. (2020, January 31). *Understanding adaboost for decision tree*. Medium. Retrieved April 17, 2022, from https://towardsdatascience.com/understanding-adaboost-for-decision-tree-ff8f07d2851
21. Scikit-learn Support Vector Machine Documentation: https://scikit-learn.org/stable/modules/svm.html

22. Witten, I. H., Frank, E., & Hall, M. A. (2011). Chapter 6.4 Extending Linear Models. In Data Mining: Practical Machine Learning Tools and Techniques. essay, Elsevier.
23. *Scikit-learn Naive Bayes Classifier*: https://scikit-learn.org/stable/modules/naive_bayes.html
24. Learn naive Bayes algorithm: Naive Bayes classifier examples. Analytics Vidhya. (2021, August 26). Retrieved May 1, 2022, from https://www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/
25. *Google Colaboratory*: https://colab.research.google.com/
26. *A STUDY ON MACHINE LEARNING ALGORITHM FOR ENHANCEMENT OF LOAN PREDICTION* by Prateek Dutta
27. *A survey on Ensemble Model for Loan Prediction* by Anchal Goyal [1], Ranpreet Kaur [2]
28. *Study on a prediction of P2P network loan default based on the machine learning LightGBM and XGboost algorithms according to different high dimensional data cleaning* by Xiaojun Maa, Jinglan Shaa, Dehua Wangb, Yuanbo Yuc, Qian Yanga, Xueqi Niu
29. Murphy, K. P. (2021). Chapter 5.7. In *Machine learning: A probabilistic perspective*. essay, MIT Press.
30. *Scikit-learn MinMaxScaler*: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

31. *Scikit-learn GridSearchCV*: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
32. *Scikit-learn 3.2.1 Exhaustive Grid Search*: https://scikit-learn.org/stable/modules/grid_search.html
33. *Scikit-learn Decision Tree Classifier*: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html
34. Decision Trees: Gini vs Entropy, by Pablo Aznar, December 2020, https://quantdare.com/decision-trees-gini-vs-entropy/
35. *Scikit-learn Random Forest Classifier*: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
36. *Scikit-learn AdaBoost Classifier*: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html
37. *Scikit-learn Support Vector Machine Classifier*: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
38. *Scikit-learn Gaussian Naive Bayes*: https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html
39.