University of Malta

Department of Artificial Intelligence

Bankruptcy Prediction

Applied Machine Learning (ICS5110) Assignment

Dylan Vasallo

dylan.vassallo.18@um.edu.mt

Peter Incorvaja

peter.incorvaja.13@um.edu.mt

January, 2019

Chapter 1: Introduction

All the scripts written for this project can be found in *'src/'* path. The code is well commented and can be easily followed. The main file for this project is the Jupyter Notebook called **bankruptcy_predictions.ipynb** which utilizes the scripts written for this project. The Notebook is also found in the *'src/'*, together with the anaconda environment used in the project which is saved as a *'.yaml'* file. Everything is publicly available on the following GitHub Repository: https://github.com/achmand/Polish-Bankruptcy-Prediction.

The dataset chosen for this project is about bankrupt Polish companies [1] that were analysed between 2000 and 2012, while the non-bankrupt companies were investigated between 2007 to 2013. The data was acquired from EMIS [2] which is a service that offers information on emerging markets. Furthermore, this dataset will be used to classify and predict whether Polish companies will go bankrupt or not.

1.1 Dataset Properties

As discussed, the dataset consists of data about Polish companies, so the domain which will be explored is Finance and Business. The total number of instances inside this dataset amount to 10503, with a total of 64 attributes/features. Each instance is labelled as non-bankrupt or bankrupt which are encoded as 0 or 1. The features inside the dataset are synthetic features meaning they are made up of different econometric measures using arithmetic operations [3]. This was done to combine econometric indicators which are well known by domain experts to create complex features which are shown in Figure 1.1. Each feature is made up of real-valued variables. The dataset is also split into different forecasting periods which are as follows;

- ***1st year***: the data contains financial rates from 1st year of the forecasting period and corresponding class label that indicates bankruptcy status after 5 years.
- ***2nd year***: the data contains financial rates from 2nd year of the forecasting period and corresponding class label that indicates bankruptcy status after 4 years.
- ***3rd year***: the data contains financial rates from 3rd year of the forecasting period and corresponding class label that indicates bankruptcy status after 3 years.
- ***4th year***: the data contains financial rates from 4th year of the forecasting period and corresponding class label that indicates bankruptcy status after 2 years.
- ***5th year***: the data contains financial rates from 5th year of the forecasting period and corresponding class label that indicates bankruptcy status after 1 year.

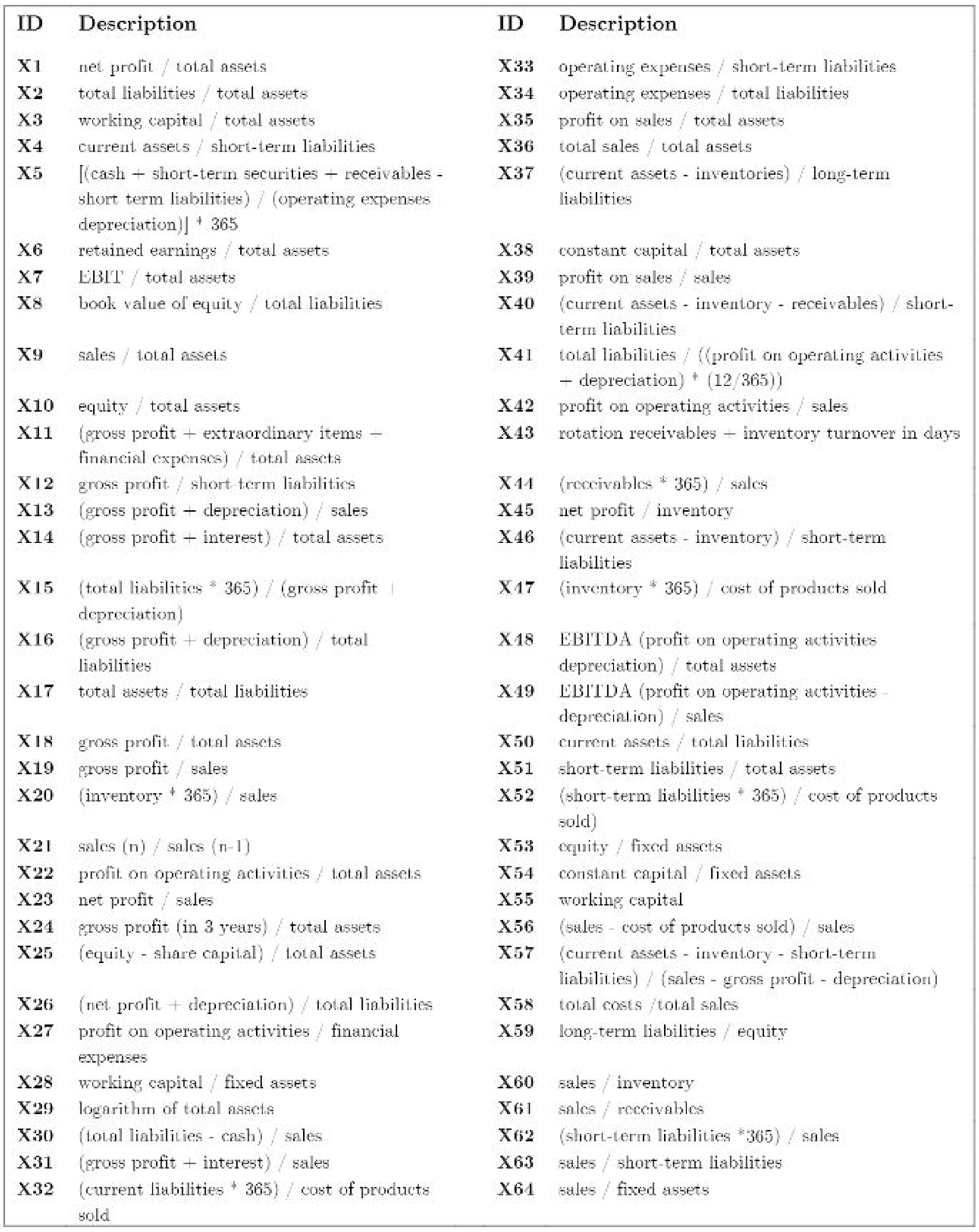
1.1.1 Feature Exploration

1.1.1.1 Missing Values

It was known that the dataset contained some missing values, so further analysis was conducted on each feature to identify these missing values. A python script called **dataexp.py** was created to analyse these features. First off, the whole dataset was checked to see what would happen if the instances with missing values were to be removed. In Table 1.1 it is shown that there would be too much data loss if these instances were to be dropped (around 50% would be lost).

Year	Total Instances	Missing Values	No Missing Values	Data Loss
1_year	7027	3833	3194	0.5455
2_year	10173	6085	4088	0.5982
3_year	10503	5618	4885	0.5349
4_year	9792	5023	4769	0.5130
5_year	5910	2879	3031	0.4871

Table 1.1: Missing values stats, output generated using **dataexp.py**



ID	Description	ID	Description
X1	net profit / total assets	X33	operating expenses / short-term liabilities
X2	total liabilities / total assets	X34	operating expenses / total liabilities
X3	working capital / total assets	X35	profit on sales / total assets
X4	current assets / short-term liabilities	X36	total sales / total assets
X5	[(cash + short-term securities + receivables - short term liabilities) / (operating expenses depreciation)] * 365	X37	(current assets - inventories) / long-term liabilities
X6	retained earnings / total assets	X38	constant capital / total assets
X7	EBIT / total assets	X39	profit on sales / sales
X8	book value of equity / total liabilities	X40	(current assets - inventory - receivables) / short-term liabilities
X9	sales / total assets	X41	total liabilities / ((profit on operating activities + depreciation) * (12/365))
X10	equity / total assets	X42	profit on operating activities / sales
X11	(gross profit + extraordinary items + financial expenses) / total assets	X43	rotation receivables + inventory turnover in days
X12	gross profit / short-term liabilities	X44	(receivables * 365) / sales
X13	(gross profit + depreciation) / sales	X45	net profit / inventory
X14	(gross profit + interest) / total assets	X46	(current assets - inventory) / short-term liabilities
X15	(total liabilities * 365) / (gross profit + depreciation)	X47	(inventory * 365) / cost of products sold
X16	(gross profit + depreciation) / total liabilities	X48	EBITDA (profit on operating activities depreciation) / total assets
X17	total assets / total liabilities	X49	EBITDA (profit on operating activities - depreciation) / sales
X18	gross profit / total assets	X50	current assets / total liabilities
X19	gross profit / sales	X51	short-term liabilities / total assets
X20	(inventory * 365) / sales	X52	(short-term liabilities * 365) / cost of products sold)
X21	sales (n) / sales (n-1)	X53	equity / fixed assets
X22	profit on operating activities / total assets	X54	constant capital / fixed assets
X23	net profit / sales	X55	working capital
X24	gross profit (in 3 years) / total assets	X56	(sales - cost of products sold) / sales
X25	(equity - share capital) / total assets	X57	(current assets - inventory - short-term liabilities) / (sales - gross profit - depreciation)
X26	(net profit + depreciation) / total liabilities	X58	total costs /total sales
X27	profit on operating activities / financial expenses	X59	long-term liabilities / equity
X28	working capital / fixed assets	X60	sales / inventory
X29	logarithm of total assets	X61	sales / receivables
X30	(total liabilities - cash) / sales	X62	(short-term liabilities *365) / sales
X31	(gross profit + interest) / sales	X63	sales / short-term liabilities
X32	(current liabilities * 365) / cost of products sold	X64	sales / fixed assets

Figure 1.1: Features in Polish Bankruptcy Dataset [3]

Furthermore, another third party library called **missinggo** [4] was used to further help us analyse/visualize these features with missing values. The nullity matrix in this library was used to find patterns visually for missing values. As shown in Figure 1.2 features **X21** and **X37** have the most missing values in 'Year 2'.

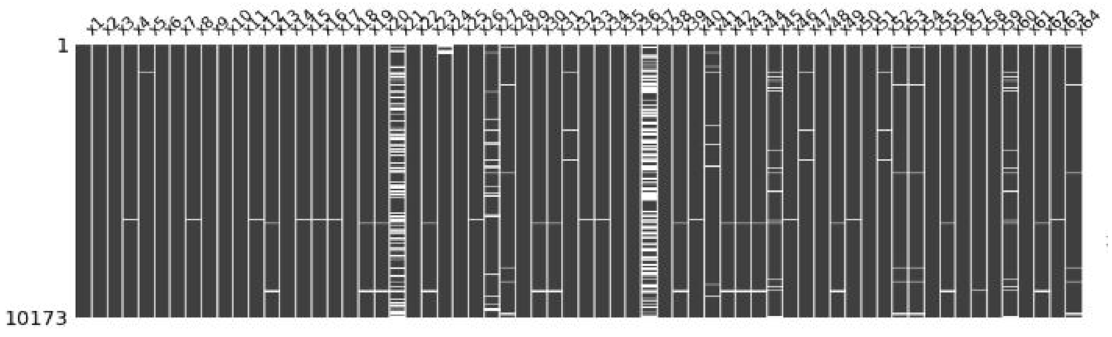


Figure 1.2: Nullity Matrix for Year 2. White space means missing values for the specific feature.

The nullity matrix plots were generated for each forecasting period/year. Another plot which was used from this library is called the correlation heatmap which measures the nullity correlation between the features. This helped to identify the correlation of nullity between each feature, meaning if a value of 1 is shown in the box, it means that if one feature is missing the other is also missing. On the other hand if -1 is shown it means if one feature is missing the other is not. The output can be a value between -1 to 1 and if the value is to small (-0.05 < V < 0.05), nothing is shown. Again these plots where generated for all forecasting periods/years. An example for this plot is shown in Figure 1.3.

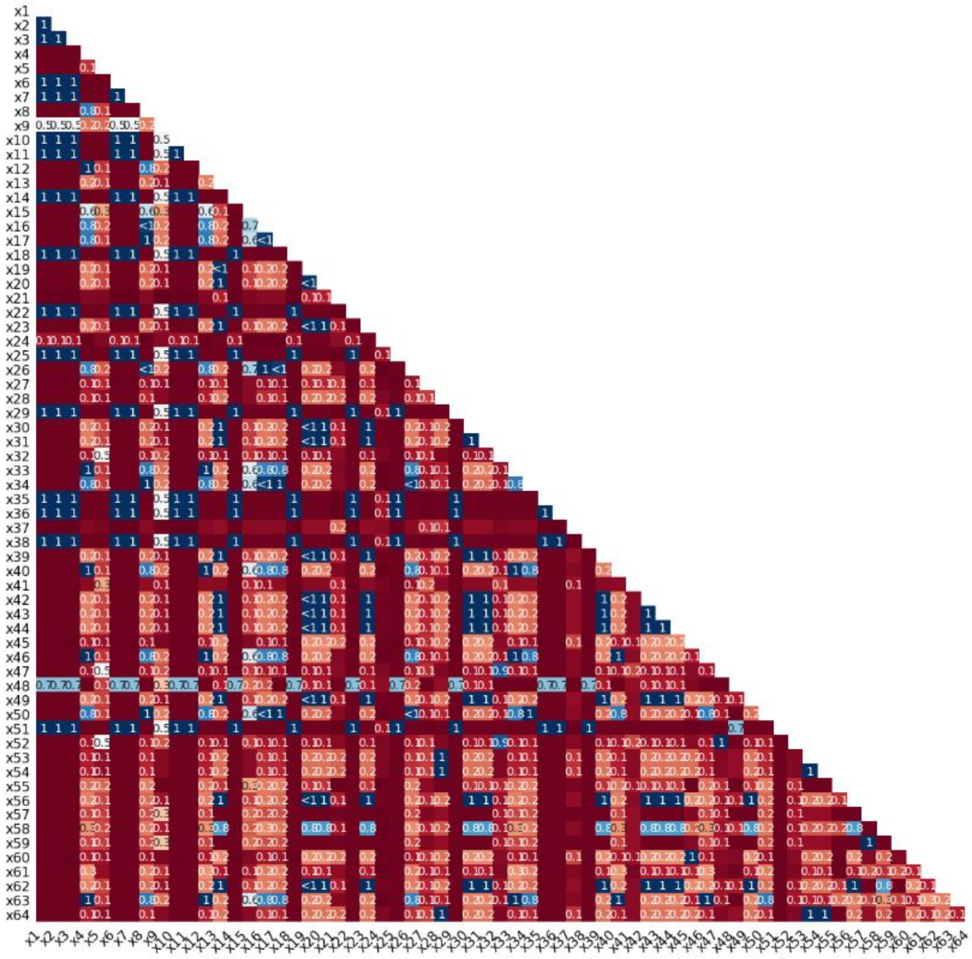


Figure 1.3: Nullity Correaltion Heatmap for Year 2

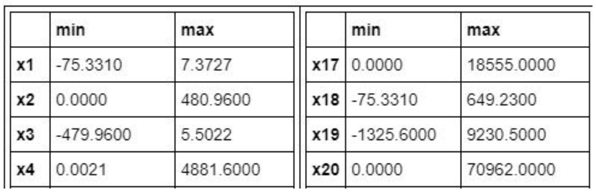
1.1.1.2 Imputation Techniques

After the missing values were analysed, it was decided that imputation techniques (replacing missing values with a substitute) would be used to fill in the missing values. Previous work done on this dataset [3] utilized the following imputation techniques; Mean, k-Nearest Neighbours, Expectation-Maximization and Multivariate Imputation by chained equations. In the experiments, Mean Imputation will be used since it produced the best results in the cited work. Moreover, Mode Imputation will also be tested to investigate whether this will be an improvement over Mean Imputation.

- ***Mean Imputation***: This technique imputes the missing data with the mean of that feature, where the data can only be numeric. Due to the nature of all the missing data being imputed with the same value, it sometimes has the undesirable effect of adding bias and reducing variance, which in turn affects the correlation value between other features.
- ***Mode Imputation***: This technique imputes the missing data with the most frequent value for that feature, with the intuition that the modal value has a higher chance of being in the dataset. The advantage of this technique is that it can also be used on non-numeric data.

1.1.1.3 Magnitude of Features

Following the analysis for the missing values, the magnitude/range for each feature was explored. For each forecasting period, the range for each feature was analysed as shown in Table 1.2. After doing so it was noted that the features contain both positive and negative values. Also the range/distance of each feature is quite sparse.



	min	max
x1	-75.3310	7.3727
x2	0.0000	480.9600
x3	-479.9600	5.5022
x4	0.0021	4881.6000

	min	max
x17	0.0000	18555.0000
x18	-75.3310	649.2300
x19	-1325.6000	9230.5000
x20	0.0000	70962.0000

Table 1.2: A snippet for the range of each feature for Year 2 using **pandas'** describe function

1.1.1.4 Distributions of Features

The distribution for each feature in different forecasting periods was plotted using a third party library called **seaborn** [5]. The 4 moments (mean, standard deviation, skew and kurtosis) were also computed and shown in the plot. This was done to check whether a feature follows a normal distribution or not as shown in Figure 1.4. The skew and kurtosis were computed using a third party library called **scipy.stats** [6]. The plots show that the features are not normally distributed (both visually and the kurtosis being far off from 3), meaning that the features are in fact not normally distributed or there are extreme outliers (heavy tails). The plots also indicate that some features are highly skewed.

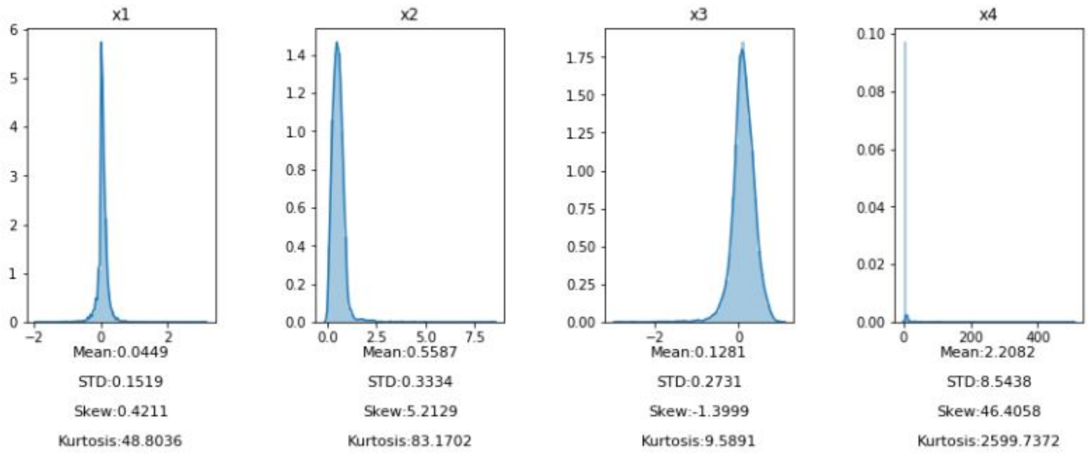


Figure 1.4: A snippet for the distributions for each feature for Year 2 using **seaborn** and **scipy.stats**

1.1.2 Data Imbalance

After exploring the features, the outcomes were analysed to check whether or not the dataset is imbalanced (labels skewed to one class). As shown in Table 1.3, the dataset is highly imbalanced with almost every forecasting period having around 96% of the instances classified as non-bankrupt.

Year	Non-bankrupt (0)	Bankrupt (1)	Minority	Minority Percentage
1_year	6756	271	1	0.038566
2_year	9773	400	1	0.039320
3_year	10008	495	1	0.047129
4_year	9277	515	1	0.052594
5_year	5500	410	1	0.069374

Table 1.3: Imbalanced labels stats, output generated using dataexp.py

This can be a problem when using classification models because the data is skewed to one label and cannot really capture how well it did when predicting whether a company will go bankrupt or not. Using this dataset without tackling this issue can result in good accuracy (other metrics can be used to capture how well the model is doing when predicting the minority class) since most instances fall under the same class, but again it won't be trained well to be able to predict the minority class (bankrupt).

To tackle such issue, synthetic data will be generated to oversample the dataset with synthetic instances belonging to the minority class. This technique will balance out the dataset and will allow the model/s to be exposed to a greater number of instances with the minority class, thus this will be improving the chances for the model to predict the correct classification when an instance is labelled as bankrupt.

1.1.2.1 Synthetic Minority Over-sampling Technique

There are various algorithms one can use to generate synthetic data. For this project, SMOTE algorithm will be used to over-sample the dataset as it is one of the most well-known technique for oversampling. SMOTE or Synthetic Minority Over-sampling Technique [7] works by selecting/sampling similar instances of the minority class, by finding the nearest neighbours k (using some distance metric based on the feature set) and changing each attribute one at time by multiplying each x by a random number (between 0 to 1), therefore creating a new synthetic instance.

1.1.3 Tackling Issues

As described in the previous Subsections 1.1.1 and 1.1.2, it was found that when analysing the properties of the dataset. So, to recap the issue of having missing values in the dataset will be tackled by using Mean and Mode imputation as described in Subsection 1.1.1.2. These imputation techniques will also be applied for comparative reasons. As for data imbalance, SMOTE technique will be used to oversample the dataset for each forecasting period/year as described in Subsection 1.1.2.1. Furthermore, other techniques will also be applied to experiment with the feature set, but these techniques will be described later on.

1.2 Machine Learning Models

For this problem three supervised machine learning algorithms will be used for classification, which are as follows; Decision Tree, Random Forest and Logistic Regression.

The Decision Tree model was firstly chosen because it supports classification. Such model supports both discrete and continuous data as features, and since the dataset contains continuous data only the Decision Tree model will work well. It supports multi class classification but for this problem only binary classification is needed. Another reason why the Decision Tree model was chosen for this project is that it is not affected by the magnitude/ranges of the features as shown is Subsection 1.1.1.3. It also does not assume that the features are normally distributed, since our dataset as shown in Subsection 1.1.1.4 does not follow a normal distribution.

Random Forest was chosen to be implemented and compared, since the Decision Tree model is susceptible to overfitting. This model will help with low bias and high variance found in Decision Trees when applied to a large dataset, with the use of ensemble techniques such as random sub space method and dataset bagging. Again, since Random Forest use an ensemble of Decision Trees (weak learners), it may help with the performance and again such model is not affected by the magnitude/range or assumes a normal distribution of the features.

Lastly the Logistic Regression model will be used in the experiments since it is a binary classification model and due to dichotomous nature of the labels found in the dataset. Similar to the decision tree, this model does not assume a normal distribution in the features. Unlike the Decision Tree model, the Logistic Regression can be affected by the magnitude of the features when training the model (adjusting weights/gradient descent). The magnitude of the features can be normalized by mapping them to a specific range, which is usually set between 0.0 to 1.0. All the models described above assume that the dataset has balanced outcomes, since it may be subjected to outcome bias and will not perform well in unforeseen instances (minority class). SMOTE technique described in Subsection 1.1.2.1 can used to oversample the minority class.

Chapter 2: Background

2.1 Selected Machine Learning Techniques

2.1.1 Decision Tree

Decision trees is a model which is a widely used technique in machine learning as it is quite simple but very powerful. This model can be used for both regression and classification problems, and there are different types of decision trees, some of which are; ID3, C4.5, CART, CHAID and MARS. In this section the CART algorithm will be discussed (as a classification problem) since this is the implementation chosen for the experiments, but the concept behind this implementation is very close to the other algorithms (the main difference in these implementation is the way the split is handled).

CART (Classification and Regression Tree) [8] is a binary tree which is built by taking a set of instances (training set) and passing them to the root node, then a decision is made (if statement) based on the features, so to partition/split the instances into two subsets and passing them as an input to the child nodes. This procedure is then applied to each node until it reaches a subset with the purest (no mixed classes) distributions of the classes, this is also known as the leaf node. The most important aspect of this procedure is to know which decision should be taken at each node to decrease the uncertainty (minimizing mixing of classes) of the subset. Firstly, a metric to quantify the uncertainty/purity at each node is needed and in this implementation Gini Impurity [9] is used as shown in Equation (2.1). The formula takes the node's class of each instance in the subset, for that node t, and computes the probability of obtaining two different outputs from all the possible classes k. If the subset only contains instances with the same label, the subset is pure, meaning $I(t) = 0$. There are other methods which can be used to quantify the impurity at each node such as Entropy, MSE and MAE, but it depends on the problem being tackled (classification or regression).

$$I(t) = 1 - \sum_{j=1}^{k} p^2(j|t) \tag{2.1}$$

$$I_g(t) = I(t) - \frac{N_{tR}}{N_t} I(t_R) - \frac{N_{tL}}{N_t} I(t_L) \tag{2.2}$$

As described above the way a node's uncertainty is measured is defined, but another metric is needed to quantify how much a certain decision reduces uncertainty, and for this Information Gain is used. Firstly, lets define what a decision is, a decision is simply a condition based on a specific feature and it usually is just a $\geq$ check when it's numeric or $==$ check when it's categorical. The Equation (2.2) shows how information gain is computed by taking the $I(t)$ of the current node and subtracting it by the total number of samples in the right node over the total number of samples in the parent node $\frac{N_{tR}}{N_t}$ multiplied by the right node uncertainty $I(t_R)$ and subtracting again with weighted average of left node uncertainty $\frac{N_{tL}}{N_t} I(t_L)$ based on a specific decision.

So, at each node every feature of each instance is checked using the Information Gain to find the best split, and once this is found the node is split into two child nodes taking as input the subset which met the criteria and the subset which did not meet the criteria. This is done recursively until there is no more further splits and by then the leaf nodes are reached. Once the tree is built/trained, one can predict the class of unforeseen instances by passing the test dataset to the tree and it will follow down the tree taking the left or right nodes based on the criteria of the decision. Once a leaf node is reached the tree will predict the probability of that instance belonging to a specific class (the highest probability is outputted if there are multiple labels in the leaf node). A maximum depth for the Tree can be set or otherwise the Tree will keep on finding the best splits until all leaves are pure (if possible).

2.1.2 Random Forest

Random forest is an ensemble of Decision Trees [10] which were described in Subsection 2.1.1. This technique utilizes multiple Decision Trees (weak learners) known as estimators to average out the predictions made by each tree, and this is done to reduce overfitting and to reduce the low bias, high variance trade-off found in a Decision Tree. There are many variations for the Random Forest implementation but it depends on the dataset being used. The bagging technique is usually used when the dataset has low bias and high variance. On the other hand, boosting is used when having high bias and low variance. In the end this aims to reduce certain undesirable implications found in a Decision Trees due to its properties.

Random Forests by their default nature, use a technique called random subspace method (Feature Bagging), which essentially selects random features, and each Decision Tree in the forest is trained using these randomly selected features. What this mean is that for each Decision Tree only a percentage of the features are randomly selected to be used in the training phase of each Decision Tree. This will reduce the correlation between each Decision Tree. The number of features to be selected randomly can be computed by; using a fixed number (smaller than the total number of features), a percentage of the total amount of features or by taking the square root, or log base 2 of the total amount of features. Furthermore, this can be also applied to bag the instances in the dataset. Meaning the full dataset is not passed to each estimator, but a subset of the dataset is used instead. Bagging the dataset is done with replacement. The total number of instances to be passed to each estimator can be computed using the same techniques when finding the total amount of features to pass in the random subspace method. Again, this will further reduce correlation between each Decision Tree.

The other method which is used is called Boosting, and there are several variations of this technique, but for the purposes of this project ADA Boost will be described. Boosting is basically a variation on Bagging and it is used when the underlying Decision Trees are not performing well as described above. In this technique a subset of the dataset is used to train the first Decision Tree and once this process is done, all the dataset is passed to the Decision Tree to validate each instance. At this point some instances will be predicted with significant error. After this the second Decision Tree will be trained and again a random subset is chosen for training but with the exception that now the instances with significant error have a higher probability to be selected (instances are weighted according to the error). Again, the second Decision Tree is validated on the whole dataset. This procedure will go on for the number of Decision Trees to be used in the Random Forest and is done sequentially. This technique is mainly used when the dataset is not complex and have fewer number of instances so underfitting may be present.

In the end when using Bagging the average of all the Decision Tree's outcome is taken if it is a regression problem, but if it is a classification problem the majority vote is taken as a final output/prediction as shown in Figure 2.1. On the other hand, in boosting, the weighted summation of each Decision Tree is taken as the final prediction.

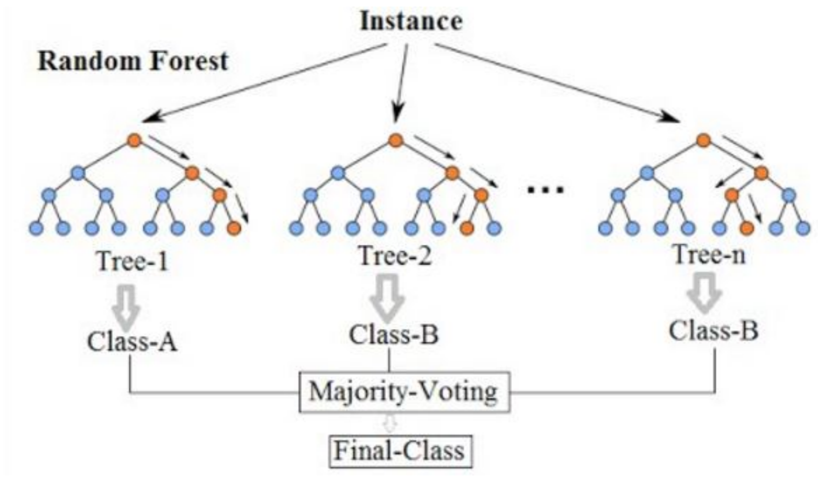


Figure 2.1: Random Forest using Bagging for a classification problem

2.1.3 Logistic Regression

Logistic regression [11] is a well-known machine learning technique used in classification problems. This model can be implemented when classifying instances [12] into dichotomous outcome like the problem being tackled in this project. Moreover, this model is also used in classification problems with non-binary outcomes [13] and it is usually referred to as multinomial or “one vs rest” Logistic Regression. Although this model can be tweaked to classify multiple classes, in this project it will be utilized to classify a binary outcome (bankrupt or non-bankrupt). In this model the Logistic Sigmoid Function (2.3) is used as a Hypothesis Function (2.4) to map an instance/s denoted as x, with some given weights denoted as θ, to its Estimated Probability (2.5) of the discrete outcome. Therefore, this function will output $0 \leq h_\theta(x) \leq 1$ as it is shown in Figure 2.2.

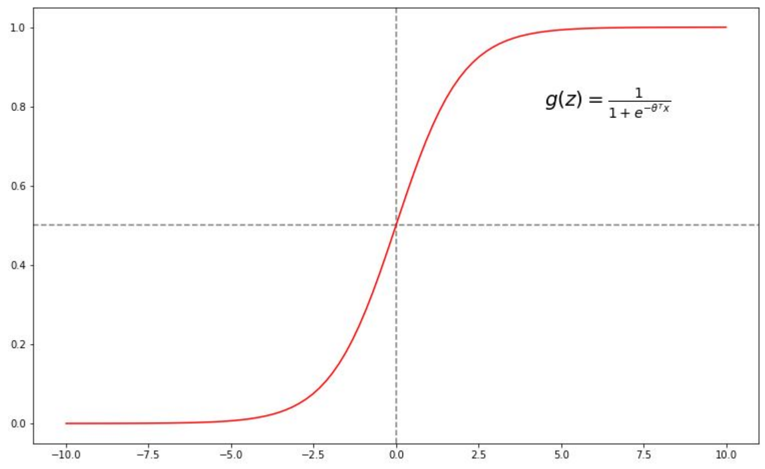


Figure 2.2: Logistic Sigmoid Function

Basically, the Hypothesis Function (2.4) will classify an instance/s to its predicted outcome. Now since the function will output the estimated probability of the discrete outcome, the model needs to interpret this output to be able to classify into 0 or 1. To do so a decision boundary is needed to predict $\hat{y} = 0 \; or \; \hat{y} = 1$. In Condition (2.6) it is shown that $\hat{y} = 1$ if $h_\theta(\theta^T x) \geq 0.5$ and $\hat{y} = 0$ if $h_\theta(\theta^T x) < 0.5$. Although in most cases the threshold is set to 0.5 as explained in Condition (2.6), in other cases this may have to be adjusted to get better results depending on the problem being tackled (threshold must always be a value between 0 and 1). The decision boundary is a property of the hypothesis, so you can have non-linear decision boundaries by adding higher polynomial terms to the features.

$$g(z) = \frac{1}{1+e^{-z}} \tag{2.3}$$

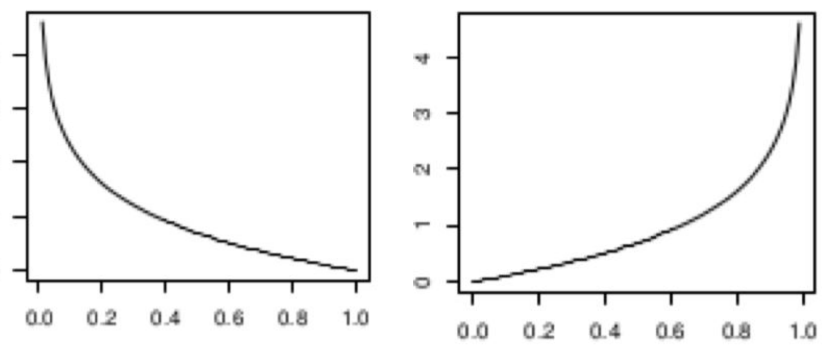
$$\begin{gathered} h_\theta = g(\theta^T x) \\ h_\theta = \frac{1}{1+e^{-g(\theta^T x)}} \end{gathered} \tag{2.4}$$

$$\begin{gathered} P = (\hat{y} = 0|x;\theta) + P = (\hat{y} = 1|x;\theta) = 1 \\ P = (\hat{y} = 0|x;\theta) = 1 - P = (\hat{y} = 1|x;\theta) \end{gathered} \tag{2.5}$$

$$\hat{y} = \begin{cases} 1 & \sigma(\theta^T x) \geq 0.5; \theta^T x \geq 0 \\ 0 & \sigma(\theta^T x) < 0.5; \theta^T x < 0 \end{cases} \tag{2.6}$$

Now that the hypothesis is defined, and the classification function is explained the model must be trained to adjust its weights θ to minimize the cost. The cost function $J(\theta)$ utilized in Logistic Regression is shown in Equation (2.7) and this cost function is used over the squared cost function to be able to find the global minimum when applying gradient descent (since it's a convex cost function when using the Logistic Sigmoid Function). A desirable property of $J(\theta)$ is that it greatly penalizes wrong predictions which have high probability with a high cost as shown in Figure 2.3.

$$\begin{gathered} Cost(h_\theta(x), y) = \begin{cases} y = 1 & -\log(h_\theta(x)) \\ y = 0 & -\log(1 - h_\theta(x)) \end{cases} \\ J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x_i), y_i) \\ J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} y_i \log h_\theta(x_i) + (1 - y_i)\log(1 - h_\theta(x_i)) \end{gathered} \tag{2.7}$$



(a) y = 1 (b) y = 0

Figure 2.3: Logistic Regression cost function

Then the weights θ are adjusted to minimize the cost function $J(\theta)$ using gradient descent as shown in Equation (2.8), until a termination condition is met. The termination condition is usually set to terminate when the number of max epochs is reached or when the step size ϵ (the difference in cost after one epoch) has reach a certain threshold. The α parameter usually referred to as the learning rate is the gradient decent step rate for each iteration which can be fixed or varied to control the steps at each iteration.

$$\begin{gathered} {}^{minimize}_{\theta} J(\theta) \\ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \end{gathered} \tag{2.8}$$

In the training phase the gradient descent equation is applied for every weight θ_j including the bias θ_0 until it converges or a termination condition is met, thus it will be adjusting weights θ_j to minimize the cost $J(\theta)$. There are other techniques which can be used to minimize $J(\theta)$ such as Newton-Raphson, Conjugate Gradient, BFGS and L-BFGS. Once the model is trained it uses the hypothesis/classification function with the adjusted θ_j to classify/predict unforeseen instances.

Regularization is sometimes used to decrease the complexity, where the most commonly used regularization methods being Lasso/L1, and Ridge/L2 regularization. In Lasso Regularization a penalty term which sums up the magnitude of all coefficients except the bias θ_0, and then scaled by parameter λ is added to $J(\theta)$ as shown in Equation (2.9). In Ridge Regression a penalty term which sums up the square of all coefficients except the bias θ_0, and then scaled by parameter λ is added to $J(\theta)$ as shown in Equation (2.10).

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} y_i \log h_\theta(x_i) + (1 - y_i) \log(1 - h_\theta(x_i)) + \underbrace{\lambda \sum_{j=1}^{n} |\theta_j|}_{\text{L1}} \tag{2.9}$$

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} y_i \log h_\theta(x_i) + (1 - y_i) \log(1 - h_\theta(x_i)) + \underbrace{\lambda \sum_{j=1}^{n} \theta_j^2}_{\text{L2}} \tag{2.10}$$

2.2 Re-scaling and Normalisation

2.2.1 Normalisation

Normalisation is a pre-processing step needed for some of the algorithms in machine learning to work or improve performance. Some examples are KNN (features having the same magnitude for distance metrics), PCA for feature reduction (when using correlation matrix), Chi2 for feature selection (doesn't work with negative values so normalisation can rescale to positive values) or when using gradient descent (makes minimizing the cost more efficient). Equation (2.11) is used to rescale values to a specific range. It describes how a value can be rescaled within a specific range x_{min} to x_{max} (commonly set to 0.0 to 1.0). When you apply this rescaling technique to all set of features (using the same range), normalisation is achieved, and all features are in the same range and as shown in Subsection 1.1.1.3 the features in our datasets contain different magnitudes and negative values.

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{2.11}$$

In this description of normalisation, it is described as having different types of values within the same range although some may use this definition to describe the process of roughly transforming a set of values to a normal distribution such a Box-Cox transformation [14] or log transformation. In the pre-processing steps, normalisation to transform to normal distribution won't be used since the datasets is highly skewed and normal distribution won't be achieved even if well-known techniques were applied. Furthermore, rescaling all features to have the same set of ranges (0 to 1) will be used to apply feature selection techniques which require positive values and for Logistic Regression (for gradient descent as described above).

2.2.2 Standardization

Standardizing the data is the process of rescaling features so to have a mean of 0 and standard deviation of 1. Unlike rescaling the data is not bound by a specified range. It is also used when the data has varying scales. This technique is more effective when applied to data that follows a normal distribution but can also be used if it does not. Standardization is commonly used in PCA with covariance matrix since you want to find a way of maximizing the variance. Standardization is calculated using the Equation (2.12), which subtracts x with the mean, and then divided with the standard deviation.

$$x_{new} = \frac{x - \mu}{\sigma} \tag{2.12}$$

2.3 Cross Validation

Validation is a technique used to evaluate and validate a model's performance. The main concept behind the validation process is to partition the dataset into a training set, which is used to train the model, and testing set, which is used to test the model. In validation the training set is only utilized once just to train the model. When this technique is used, one must specify the percentage of the split which is usually set between 60:40 ratio or 70:30 ratio.

The problem with splitting the dataset into two partitions is that the model will only give as how well it performed based on the data that was used to train it. Now this could lead to overfitting or underfitting the dataset and may not give a good indicator of how it will perform in a more generalized way (unforeseen data outside these two partitions). Validation also tends to induce some testing bias since we reserved a piece of the dataset just for testing.

To tackle such problems and fully utilize our dataset, cross-validation can be used. There are various techniques which try to ensure low bias and low variance such as K-Fold Cross Validation, Stratified K-Fold Cross Validation and Leave-P-Out Cross Validation. In this section we will be discussing K-Fold cross validation [15] since it was decided to use this method in our experiments, as it is one of the most commonly used method for cross validation.

In this technique the dataset available is split into k partitions, and k-1 partitions is used as a training set and the remaining partition is used as a testing set once. Then this process is iterated for k number of times, so in the end of the iterations every partition will be used once as a testing set. At each iteration/fold some performance metrics are measured and by the end of the iterations these metrics are averaged out. This reduces both bias and variance as the original dataset is used for both training and testing sets, and by doing so the model is neither overfitted due to having 1 large training set, nor is it being underfitted due to having a larger test set than the traditional validation method. The dataset prior to splitting can remain either contiguous or randomized.

Moreover, this technique can be applied to multiple models to help in the selection of the best performing model by using the same folds for each model at each iteration. This also enables the facility to tune hyper-parameters by using this cross-validation technique with the same model but different hyperparameters and selecting the best at the end of the iteration (using some performance metrics). The parameter k is usually set to $k = 5$ or $k = 10$ [16] as there is a bias-variance trade-off when choosing k and these are known to produce a good balance between the two.

2.4 Dimensionality Reduction and Feature Selection

Feature selection techniques are methods used to eliminate features which are redundant or irrelevant. This is achieved using different methods but the most common is to find any features which are correlated to each other (or correlation with the outcome) thus removing redundant and unwanted features. This in turn can simplify the hypothesis in a model which can improve both performance and memory allocation, while also reduces overfitting (reducing bias). There are various techniques to achieve this but for the purpose of this project, Recursive Feature Elimination (RFE) and Chi2 feature selection will be described.

On the other hand, dimensionality reduction, reduces the dimensionality by creating new synthetic features based on the original features using some method unlike feature selection which chooses a subset of the original features. This is done by reducing multiple correlated features/dimensions into fewer synthetic features/dimensions, like compressing 2 dimensions into 1 dimension. This has the same ramifications as feature selection (performance and memory) but except for trying to keep as much information as possible. It is also used to help visualize multi-dimensional datasets in plots.

2.4.1 Recursive Feature Elimination

The Recursive Feature Elimination [17], as the name suggests, recursively fits the model and eliminates the least important feature/s with every iteration. This is done by ranking the features according to their coefficient weight and eliminating the least weighted feature/s, depending if the algorithm is set to eliminate 1 or more feature with every iteration. After each iteration the model is fitted again, and the least weighted feature/s are eliminated again, until the specified number of maximum iterations or features to be eliminated is reached. In this project, this will be used with the Logistic Regression classifier.

2.4.2 Chi2 Feature Selection

The Chi2 test [18] is used to determine if the feature/s and outcome are dependent or not. It is important that the features and classes do not contain non-negative values (re-scaling can be used to change negative values to positive values, by changing the range of the values as described in Subsection 2.2.1). It is often used in Machine Learning to rank features based on their Chi2 statistic (score). Features which are not found to be dependent on the outcome (irrelevant for classification), are discarded depending on a specified number of features to be selected, so that only the top ranked features are selected.

2.4.3 Principal Component Analysis

Principal Component Analysis [19], or PCA for short, is a dimensionality reduction technique where its main objective, as the name suggests, is to identify principal components in a dataset. Principal components are identified with the help of a covariance matrix as shown in Equation (2.13) of the dataset. When two or more variables have a high positive covariance, the information in both can be represented by a single Principal Component.

$$\Sigma = \sum_{i=1}^{n}(x_i)(x_i)^T \tag{2.13}$$

$$\Sigma = U \cdot S \cdot V^T \tag{2.14}$$

After calculating the covariance matrix, the eigenvectors and their eigenvalues are calculated by decomposing the covariance matrix. This is done by several methods, but the most commonly used method is Singular-Value Decomposition, or SVD for short. What SVD does as shown in Equation (2.14), is decompose $m \times n$ into 3 matrices, $m \times m$ unitary matrix U, an $m \times n$ diagonal matrix S and an $n \times n$ unitary matrix V. From this decomposition, the eigevalues and eigenvectors are extracted. This process have been simplified for the purposes of this project.

The eigenvectors represent the direction of the new axis' and the eigenvalues represent the variance of data on that eigenvector. The eigenvectors are then sorted by their eigenvalues, and the eigenvectors that have the least variance can be discarded. For a k-dimensional dataset, there can only be k-eigenvectors, as all the combination of eigenvectors must span over the k-dimensional space. Therefore, when eliminating the eigenvectors that represent the data with least variance, dimensions are being reduced, as their data can still be projected with minimal error on the eigenvectors that represent the data with most variance.

2.5 Performance Metrics

In this classification problem the following performance metrics will be used to quantify the results for the different experiments conducted at a later stage;

- ***Accuracy***: The most basic metric for classification problems, which basically is the number of correct classifications (predicted) made up by the model over all the predictions (correct and incorrect) which were made by the same model.
- ***Recall***: Such metric quantifies how many actual positives were predicted by the model by classifying them as positives. In the experiments this metric will be given high importance since the cost of having a false negative is quite important in this situation meaning that the model must not classify a company as non-bankrupt when it is bankrupt. Recall is calculated using the Equation (2.15).

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \tag{2.15}$$

- ***Precision***: This metric measure how precise is the model when predicting positive values (how many of these predictions are actually positive). It will also determine whether the model is misclassifying non-bankrupt companies as positives and this is quite important in this problem since it can infer misleading information (not investing in such companies because the model predicted that a company will go bankrupt). Precision is calculated using the Equation (2.16).

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Positive} \tag{2.16}$$

- ***F1 Score***: This metric will be used to select the best model from the experiments since both precision and recall are highly important in this situation. This score is the weighted average of precision and recall. The highest score for F1 is 1 and the lowest is 0. F1 score is calculated using the following Equation (2.17).

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{2.17}$$

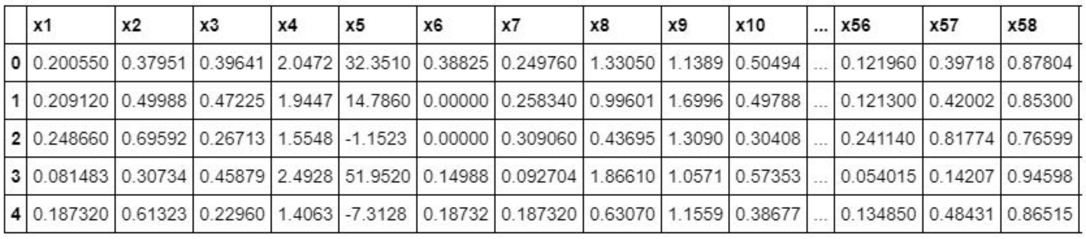
- ***Confusion Matrix***: This matrix will be used to help to visualize the variables used in the metrics described above. The matrix will output the True Positive, False Negative, False Positive and True Negative of each class. Below is an example of a confusion matrix.

		Prediction outcome		
		p	**n**	**total**
actual value	**p′**	True Positive	False Negative	P′
	n′	False Positive	True Negative	N′
	total	P	N	

Chapter 3: Experiments

3.1 Processing the Dataset

First of the dataset was loaded from *'.arff'* files using a script that was created, called **dataio.py**. Each forecasting period have its own file and these datasets can be found in *'data/'* path. A third party library called **arff** [20] was used in this script to load the data from files with an *'.arff'* format. These files were then converted into **pandas** [21] dataframes, where an example can be shown in Table 3.1.



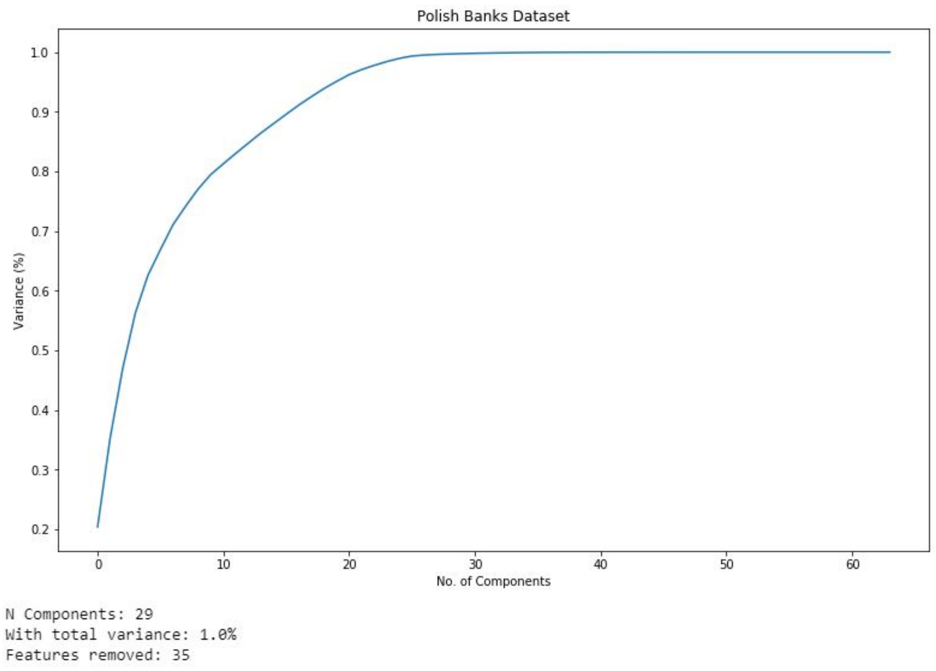
	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	...	x56	x57	x58
0	0.200550	0.37951	0.39641	2.0472	32.3510	0.38825	0.249760	1.33050	1.1389	0.50494	...	0.121960	0.39718	0.87804
1	0.209120	0.49988	0.47225	1.9447	14.7860	0.00000	0.258340	0.99601	1.6996	0.49788	...	0.121300	0.42002	0.85300
2	0.248660	0.69592	0.26713	1.5548	-1.1523	0.00000	0.309060	0.43695	1.3090	0.30408	...	0.241140	0.81774	0.76599
3	0.081483	0.30734	0.45879	2.4928	51.9520	0.14988	0.092704	1.86610	1.0571	0.57353	...	0.054015	0.14207	0.94598
4	0.187320	0.61323	0.22960	1.4063	-7.3128	0.18732	0.187320	0.63070	1.1559	0.38677	...	0.134850	0.48431	0.86515

Table 3.1: An example of a **pandas'** dataframe for Year 1

Once all the datasets were loaded, the imputation techniques (Mean and Mode) described in Section 1.1.1.2 were applied. To do so a script was created called **preprocessing.py** and this script was called from the IPython notebook to apply these techniques. In this script a third party library called **sklearn.impute** [22] was utilized to help with the imputation process. The imputation techniques were applied to each forecasting periods and once this process was done they were saved in *'data/'* path as *'.arff'* files. A function from the **preprocessing.py** script was called to do so, and this function uses a third party library **scipy.io** [6] to save these files. This was done so the imputation process would be done only once and whenever these imputed datasets are needed, they can be called from the *'data/'* folder.

After the imputation techniques were applied, oversampling using SMOTE as described in Subsection 1.1.2.1 was applied to each imputed dataset for each forecasting period. A third party library called **imblearn.over_sampling** [23] was utilized in the **preprocessing.py** script to apply oversampling. After oversampling, there was a total of 10 different datasets as **pandas'** dataframes (5 forecasting periods * 2 imputation techniques) and the number of instances almost doubled after this (refer to Table 1.3 for the original number of instances).

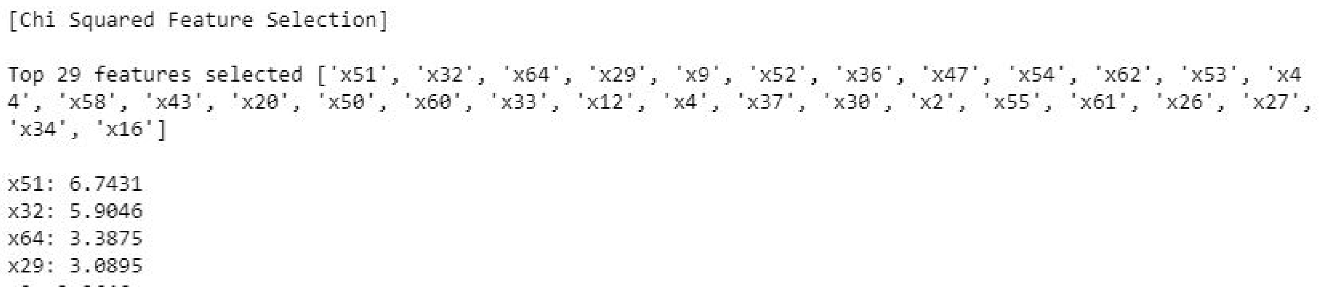
Following the oversampling process, dimensionality reduction and feature selection techniques described in Section 2.4 were applied. A script called **feature_extraction.py** was created for such processes. First an analysis was done to find the number of components to be used in PCA. The first forecasting period (1st year) in the Mean imputed dataset was used for this analysis, and this was done to get a rough estimate on the number of components to use. A copy of this dataset was standardized as described in Subsection 2.2.2, since the covariance matrix technique will be used. A plot was generated as shown in Figure 3.1 to find the best number of components which keeps as much variance as possible. The best number of components was selected which was 29 components. PCA was then applied to each imputed dataset, for each forecasting period, and the features were reduced to 29 dimensions (the original copy of the datasets were kept since they will be used in the experiments). A third party library **sklearn.decomposition** [22] was utilized in **feature_extraction.py** to apply PCA. Since the features were reduced to 29, the top 29 features will be selected in Feature Selection techniques, this is done for comparative reasons.



```
N Components: 29
With total variance: 1.0%
Features removed: 35
```

Figure 3.1: PCA analysis on Mean Imputed Year 1 forecasting period

Chi2 Feature Selection as described in Subsection 2.4.2 was then applied. Similar to the process for PCA the 1st forecasting period for the Mean Imputed dataset was used to get a rough estimate of the most important features. Since the dataset contains negative values as shown in Subsection 1.1.1.3, the features were re-scaled using the Equation 2.11 to the range between 0 and 1 (using **preprocessing.py** script). The Chi2 statistic score was computed for each feature and the top 29 features were selected as shown in Figure 3.2. A copy of the datasets with these selected features were taken to be used in the experiments. A third party library **sklearn.feature_selection** [22] was utilized in **feature_extraction.py** to apply Chi2 Feature Selection.



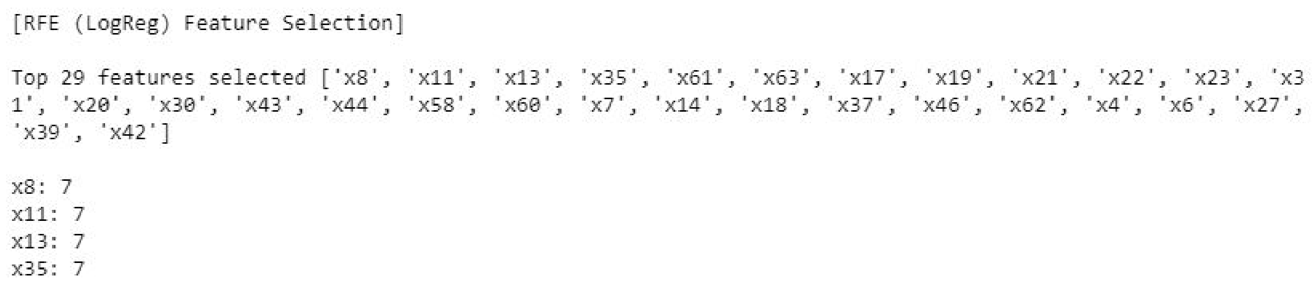
```
[Chi Squared Feature Selection]

Top 29 features selected ['x51', 'x32', 'x64', 'x29', 'x9', 'x52', 'x36', 'x47', 'x54', 'x62', 'x53', 'x4
4', 'x58', 'x43', 'x20', 'x50', 'x60', 'x33', 'x12', 'x4', 'x37', 'x30', 'x2', 'x55', 'x61', 'x26', 'x27',
'x34', 'x16']

x51: 6.7431
x32: 5.9046
x64: 3.3875
x29: 3.0895
```

Figure 3.2: Top features selected using Chi2 based on Year 1 Mean Imputed dataset

RFE technique was then applied to select the top 29 features as described in Subsection 2.4.1. Again, as the two previous techniques the two datasets was used to get a rough estimate of the most important features. The Logistic Regression model which was described in Subsection 2.1.3 was used as an estimator for this technique and a termination condition of 1000 epochs was used. For the reasons described in Subsection 2.2.1 the dataset was also re-scaled to the range of 0 and 1 to help with gradient descent. The top 29 features selected by this technique are shown in Figure 3.3. A copy of the datasets with these selected features were taken to be used in the experiments. A third party **sklearn.feature_selection** [22] was utilized in **feature_extraction.py** to apply RFE. Now that the datasets have been processed, the experiments described in the next section can be conducted.



```
[RFE (LogReg) Feature Selection]

Top 29 features selected ['x8', 'x11', 'x13', 'x35', 'x61', 'x63', 'x17', 'x19', 'x21', 'x22', 'x23', 'x3
1', 'x20', 'x30', 'x43', 'x44', 'x58', 'x60', 'x7', 'x14', 'x18', 'x37', 'x46', 'x62', 'x4', 'x6', 'x27',
'x39', 'x42']

x8: 7
x11: 7
x13: 7
x35: 7
```

Figure 3.3: Top features selected using RFE based on Year 1 Mean Imputed dataset

3.2 Carried Out Experiments

Experiments carried out can be found in the IPython notebook in the 'Perform Data Modelling' section. In total four experiments were carried out which are as follows;

- ***Experiment No 1***: Oversampled dataset for each forecasting period using different imputation techniques (Mean and Mode).
- ***Experiment No 2***: Oversampled dataset with feature reduction (PCA) for each forecasting period using different imputation techniques.
- ***Experiment No 3***: Oversampled dataset with feature selection (Chi2) for each forecasting period using different imputation techniques.
- ***Experiment No 4***: Oversampled dataset with feature selection (RFE) for each forecasting period using different imputation techniques.

The dataset was oversampled using SMOTE which was described in Subsection 1.1.2.1. Each experiment was conducted using the proposed models in Subsection 1.2 and using the imputation techniques discussed in Subsection 1.1.1.2. All experiments where validated using K-fold cross validation $k = 5$ which was described in Section 2.3 and the results were quantified using the performance metrics mentioned in Section 2.5. Furthermore, when Logistic Regression was used, the features were rescaled between the range of 0 and 1, for the same reasons described in Subsection 2.2.1. A script called **model_validation.py** was created to test different experiments.

3.3 Implementation of Machine Learning Techniques

All implementations described below were initialized in the Jupyter Notebook and validated during each experiment. Every implementation uses **numpy** [24] library for arrays and this was done to get better performance. Below each implementation is desrcibed in further detail.

3.3.1 Decision Tree

For the implementation of Decision Tree two python scripts were created **decision_tree_criterion.pyx** and **decision_tree.py**. The **decision_tree_criterion.pyx** have the following functions which were described in Subsection 2.1.1; *binary_gini_impurity*, *binary_information_gain* and *decision_split* which were later utilized in the **decision_tree.py** script. In the **decision_tree_criterion.pyx** script, **cython** [25] (python to c compiler) was utilized to achieve better performance hence the *'.pyx'* format.

Three classes can be found in the **decision_tree.py** script which are as follows; *LeafNode*, *InternalNode* and *DecisionTree*. Once a Decision Tree is initialized the *fit* function is called to start the training phase. This function takes two parameters, the instances X and their labels y. In turn, this function calls the *construct_tree* function to construct the tree using recursion. On the first iteration the *get_split* function is called which returns the best gain and decision. The returned best gain is checked and if the gain is equal to 0 a *LeafNode* is returned. If not, the instances are split based on the best decision returned and the *contruct_tree* function is called on the split nodes (hence recursion). At the end of this recursive procedure an *InternalNode* (root node) is returned with a reference to the first two nodes at the beginning of the recursion.

Moreover, a heuristic was added in the *get_split* function so not every value of each feature is checked. This was done by sorting the values of the feature being checked and only take values which have different labels in the sequence of sorted values. Then the average of every two values in the sequence is used to test the split. Once the tree is trained the *predict* function can be called to output an array of the predicted values based on the instances passed x.

The end implementation for Decision Tree used Gini Impurity to calculate uncertainty, Gini information Gain to quantify the best split using a particular feature, recursion to construct the tree and finding the best split at each node, and it keeps on adding nodes until pure leaf nodes (if possible) are found.

3.3.2 Random Forest

The code for Random Forest can be found in **random_forest.py**. This code imports **decision_tree.py** since it uses a collection of *DecisionTree* to build to model. This script has a class named *RandomForest* and it takes the following arguments in the constructor; *n_estimators* (number of Decision Trees), *max_features* (number of features to be selected randomly), *bag_ratio* (the percentage of the total number of instances to be used to bag the instances), *bag_features* (boolean to check if bagging on the dataset will be used), and *random_seed* (to seed the random instance).

When an instance of *RandomForest* is initialized the *fit* function can be called to start training the model. This function takes the instances X as a parameter and their labels y. The *fit* function then calls the *construct_forest* function, and iterates for the number of *n_estimators* passed. In each iteration the random subspace method is done based on the parameter passed as *max_features* and if *bag_features* is set to true, the dataset bagging is done based on the percentage passed in *bag_ratio*, then an instance of *DecisionTree* is initialized and it's *fit* function is called. All the initialized estimators are saved in a list. Once the training phase is completed the *predict* function can be called which takes a set of instances x as an argument and returns an array with the predicted outcomes (using the majority vote method since its a classification problem).

For our experiments Random Forest was initialized using 10 Decision Trees with the same implementation described in Subsection 3.3.1, with random subspace method (number of features selected using square root method) and dataset bagging (60% of the total instances are selected randomly and passed to each estimator) all of which were described in Subsection 2.1.2.

3.3.3 Logistic Regression

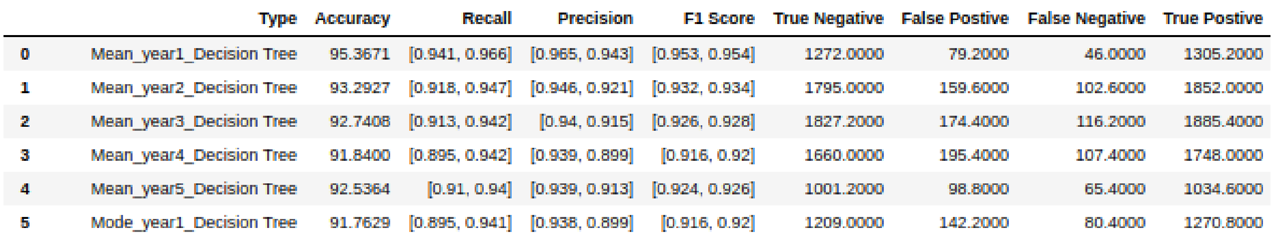
The Logistic Regression implementation can be found in the script **logistic_regression.py**. It inherits two **sklearn** [22] base classes named *BaseEstimator* and *ClassifierMixin*. This was done to integrate the Logstic Regression model with the **sklearn** RFE function as described in Section 3.1 (for Feature Selection). All the logic for the training and predictions were implemented using first principles and these inherited classes were only used so that this model can be used as an estimator in some **sklearn** functions such as RFE. The class can be initialized with the following parameters; *alpha* (the learning rate), *max_epochs* (the number of iterations), *penalty* (L1 or L2 regularization), *lambda_t* (regularization term) and *verbose* (to display stats in each iteration in Gradient Descent).

The *fit* function uses Gradient Descent to minimize J(θ), where feature coefficients vector θ values are initialized to 0, and a bias vector initialized to 1 is added to each example's coefficients. It also has the option to either use L1 or L2 regularization. To apply Gradient Descent the *fit* function loops for a specific number of epoch as passed in *max_epochs* and adjust the weights/thetas using the method described in Subsection 2.1.3, with the specified *alpha* as the learning rate for the Gradient Descent. The *predict* function predicts the outcome of a particular company's set of data, to determine if it will go bankrupt or not, by passing the features through the already trained logistic model and returning an array with the predicted classification.

In the Experiments the Logistic Regression model was initialized using an alpha of 0.01, max epochs of 1000, using Ridge Regression (L2) with regularization term lambda of 0.1.

3.4 Results from Experiments

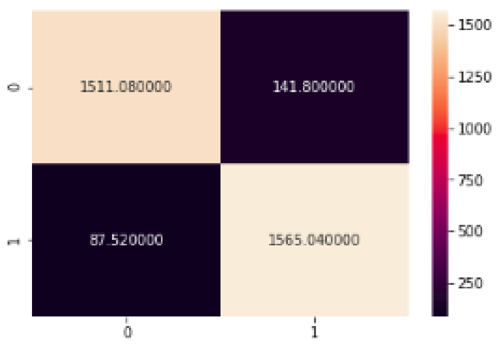
After all experiments were conducted using the implementations described in Section 3.3 the results were converted into **pandas'** dataframes and later outputted as shown in Table 3.2, with the metrics described in Section 2.5. Table 3.2 just shows a snippet of the dataframe where in total 30 rows (each model with two different imputation techniques for the five forecasting periods) were shown. This process was then done for each experiment described in Section 3.2, so in the end four **panda's** dataframes were outputted.



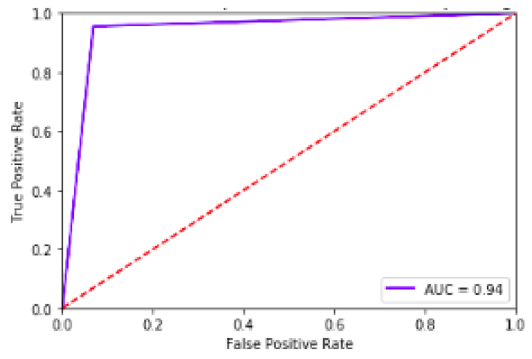
	Type	Accuracy	Recall	Precision	F1 Score	True Negative	False Postive	False Negative	True Postive
0	Mean_year1_Decision Tree	95.3671	[0.941, 0.966]	[0.965, 0.943]	[0.953, 0.954]	1272.0000	79.2000	46.0000	1305.2000
1	Mean_year2_Decision Tree	93.2927	[0.918, 0.947]	[0.946, 0.921]	[0.932, 0.934]	1795.0000	159.6000	102.6000	1852.0000
2	Mean_year3_Decision Tree	92.7408	[0.913, 0.942]	[0.94, 0.915]	[0.926, 0.928]	1827.2000	174.4000	116.2000	1885.4000
3	Mean_year4_Decision Tree	91.8400	[0.895, 0.942]	[0.939, 0.899]	[0.916, 0.92]	1660.0000	195.4000	107.4000	1748.0000
4	Mean_year5_Decision Tree	92.5364	[0.91, 0.94]	[0.939, 0.913]	[0.924, 0.926]	1001.2000	98.8000	65.4000	1034.6000
5	Mode_year1_Decision Tree	91.7629	[0.895, 0.941]	[0.938, 0.899]	[0.916, 0.92]	1209.0000	142.2000	80.4000	1270.8000

Table 3.2: A snippet for results for Experiment 1 (From First Principles)

Based on the results which were described above, the confusion matrix and ROC curve were plotted as shown in Figure 3.4, for the best performing forecasting period (1st year), model (Decision Tree) and imputation technique (Mean).



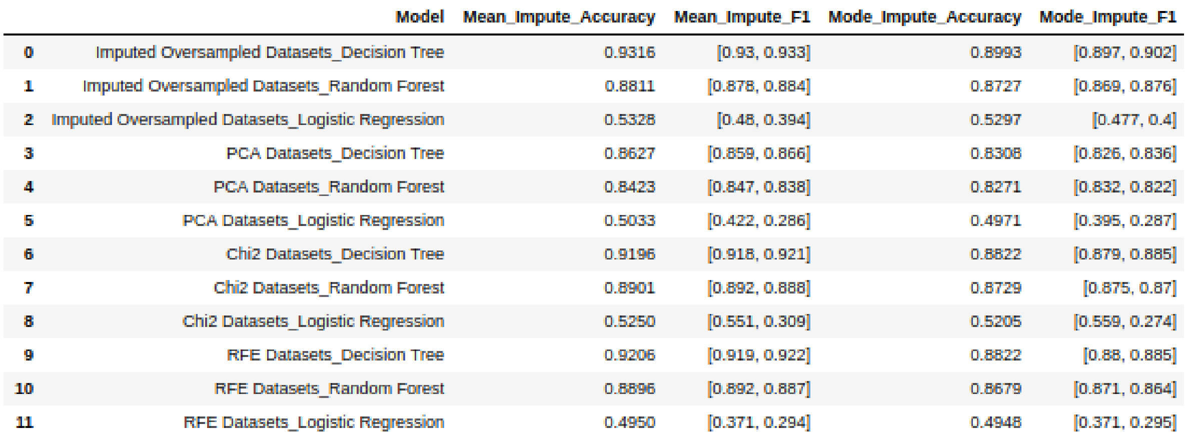
(a) Confusion Matrix



(b) ROC curve

Figure 3.4: Plots for best performing Model (Decision Tree) using best imputation technique (Mean); Confusion Matrix using averaged values of all forecasting periods and ROC curve for best forecasting period (1st Year) using first principles

Following the first step to output the results (each model using different imputation techniques for every forecasting period for four experiments), all the results were averaged by the forecasting periods to get a better metric for comparison. Table 3.3 shows the averaged results for each model and experiment.

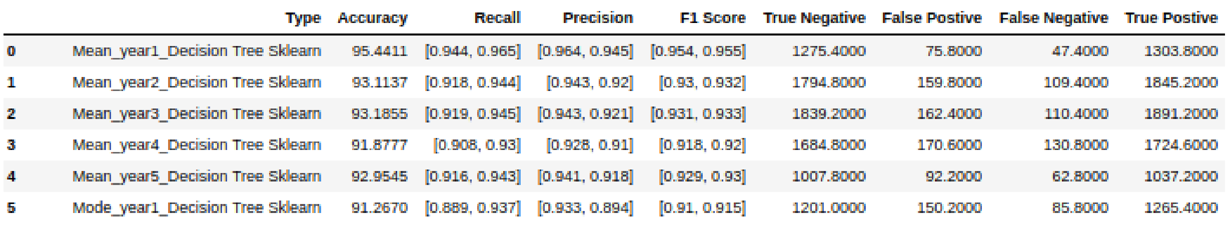


	Model	Mean_Impute_Accuracy	Mean_Impute_F1	Mode_Impute_Accuracy	Mode_Impute_F1
0	Imputed Oversampled Datasets_Decision Tree	0.9316	[0.93, 0.933]	0.8993	[0.897, 0.902]
1	Imputed Oversampled Datasets_Random Forest	0.8811	[0.878, 0.884]	0.8727	[0.869, 0.876]
2	Imputed Oversampled Datasets_Logistic Regression	0.5328	[0.48, 0.394]	0.5297	[0.477, 0.4]
3	PCA Datasets_Decision Tree	0.8627	[0.859, 0.866]	0.8308	[0.826, 0.836]
4	PCA Datasets_Random Forest	0.8423	[0.847, 0.838]	0.8271	[0.832, 0.822]
5	PCA Datasets_Logistic Regression	0.5033	[0.422, 0.286]	0.4971	[0.395, 0.287]
6	Chi2 Datasets_Decision Tree	0.9196	[0.918, 0.921]	0.8822	[0.879, 0.885]
7	Chi2 Datasets_Random Forest	0.8901	[0.892, 0.888]	0.8729	[0.875, 0.87]
8	Chi2 Datasets_Logistic Regression	0.5250	[0.551, 0.309]	0.5205	[0.559, 0.274]
9	RFE Datasets_Decision Tree	0.9206	[0.919, 0.922]	0.8822	[0.88, 0.885]
10	RFE Datasets_Random Forest	0.8896	[0.892, 0.887]	0.8679	[0.871, 0.864]
11	RFE Datasets_Logistic Regression	0.4950	[0.371, 0.294]	0.4948	[0.371, 0.295]

Table 3.3: The mean results (forecasting periods) for each Experiment (From First Principles)

3.5 Comparing to Third Party Libraries

The same experiments were conducted using **sklearn**'s implementation of these model. The same parameters were used as the first principal implementation, so to be able to compare these two. The same procedure described in Section 3.4 was done to gather results. The main difference between these two is the running time, the first principal implementations took the following time to finish; 21921 s (1st Experiment), 15762 s (2nd Experiment), 14487 s (3rd Experiment) and 13603 s (4th Experiment) while the **sklearn** implementations took; 115 s (1sth Experiment), 60 s (2nd Experiment), 59 s (3rd Experiment) and 54 s (4th Experiment). After some test it was noted that the implementation of the Decision Tree was the bottleneck in first principal implementation. This could be because no parallelization was used in the implementation and **sklearn**'s Decision Tree is fully implemented using **cython**.

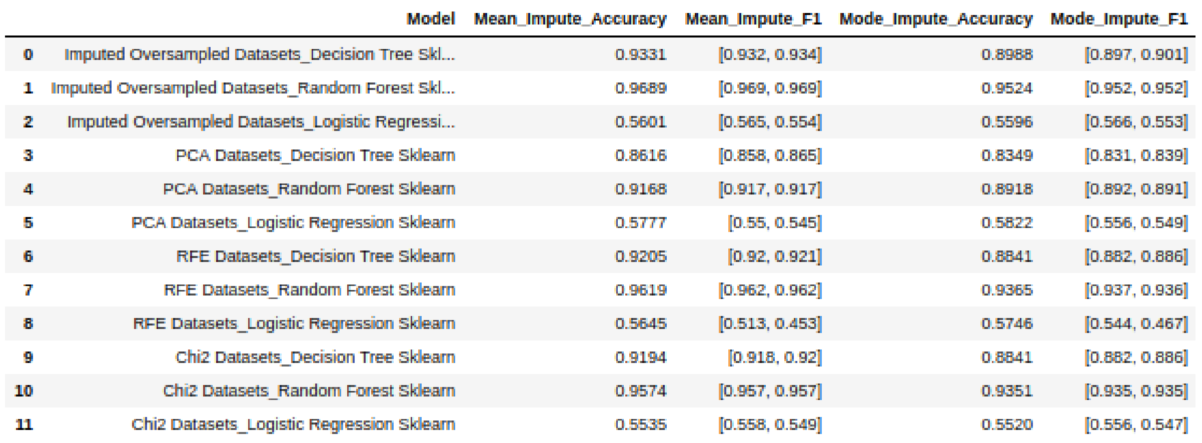


	Type	Accuracy	Recall	Precision	F1 Score	True Negative	False Postive	False Negative	True Postive
0	Mean_year1_Decision Tree Sklearn	95.4411	[0.944, 0.965]	[0.964, 0.945]	[0.954, 0.955]	1275.4000	75.8000	47.4000	1303.8000
1	Mean_year2_Decision Tree Sklearn	93.1137	[0.918, 0.944]	[0.943, 0.92]	[0.93, 0.932]	1794.8000	159.8000	109.4000	1845.2000
2	Mean_year3_Decision Tree Sklearn	93.1855	[0.919, 0.945]	[0.943, 0.921]	[0.931, 0.933]	1839.2000	162.4000	110.4000	1891.2000
3	Mean_year4_Decision Tree Sklearn	91.8777	[0.908, 0.93]	[0.928, 0.91]	[0.918, 0.92]	1684.8000	170.6000	130.8000	1724.6000
4	Mean_year5_Decision Tree Sklearn	92.9545	[0.916, 0.943]	[0.941, 0.918]	[0.929, 0.93]	1007.8000	92.2000	62.8000	1037.2000
5	Mode_year1_Decision Tree Sklearn	91.2670	[0.889, 0.937]	[0.933, 0.894]	[0.91, 0.915]	1201.0000	150.2000	85.8000	1265.4000

Table 3.4: A snippet for results for Experiment 1 (Sklearn)

When comparing the results shown in Table 3.4, both first principal implementations and **sklearn** implementations seem achieve the same accuracy and F1 scores. Both of which worked better when using Mean Imputation. Although differences were noted when comparing to the average results of the forecasting periods shown in Table 3.5. It was noted that the best model when using **sklearn** was the Random Forest using only oversampled datasets (Mean Imputation) with an accuracy of 96.89% and an F1 score of [96.9%, 96.9%]. When comparing this model to the first principal implementation one, there is a difference of +8.78% in accuracy and a difference in F1 score of [+9.1%, +8.5%].

Unlike **sklearn** implementation the best model in the first principles implementation was the Decision Tree using only oversampled datasets (Mean Imputation). But when comparing the **sklearn** Decision tree implementation using only oversampled datasests (Mean Imputation) with the first principles implementation the differences are quite minimal when compared to the Random Forest implementation. The differences being +0.15% in accurary and a difference of [+0.2, +0.1] in the F1 score. In both implementations the Logistic Regression had very poor results and in both implementations Feature Selection worked better than Feature Reduction.

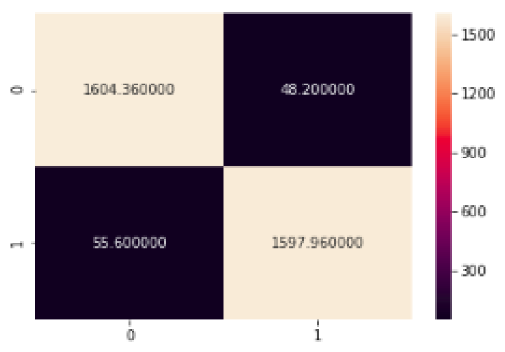


	Model	Mean_Impute_Accuracy	Mean_Impute_F1	Mode_Impute_Accuracy	Mode_Impute_F1
0	Imputed Oversampled Datasets_Decision Tree Skl...	0.9331	[0.932, 0.934]	0.8988	[0.897, 0.901]
1	Imputed Oversampled Datasets_Random Forest Skl...	0.9689	[0.969, 0.969]	0.9524	[0.952, 0.952]
2	Imputed Oversampled Datasets_Logistic Regressi...	0.5601	[0.565, 0.554]	0.5596	[0.566, 0.553]
3	PCA Datasets_Decision Tree Sklearn	0.8616	[0.858, 0.865]	0.8349	[0.831, 0.839]
4	PCA Datasets_Random Forest Sklearn	0.9168	[0.917, 0.917]	0.8918	[0.892, 0.891]
5	PCA Datasets_Logistic Regression Sklearn	0.5777	[0.55, 0.545]	0.5822	[0.556, 0.549]
6	RFE Datasets_Decision Tree Sklearn	0.9205	[0.92, 0.921]	0.8841	[0.882, 0.886]
7	RFE Datasets_Random Forest Sklearn	0.9619	[0.962, 0.962]	0.9365	[0.937, 0.936]
8	RFE Datasets_Logistic Regression Sklearn	0.5645	[0.513, 0.453]	0.5746	[0.544, 0.467]
9	Chi2 Datasets_Decision Tree Sklearn	0.9194	[0.918, 0.92]	0.8841	[0.882, 0.886]
10	Chi2 Datasets_Random Forest Sklearn	0.9574	[0.957, 0.957]	0.9351	[0.935, 0.935]
11	Chi2 Datasets_Logistic Regression Sklearn	0.5535	[0.558, 0.549]	0.5520	[0.556, 0.547]

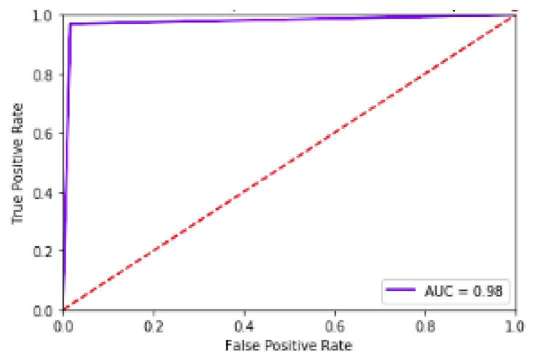
Table 3.5: The mean results (forecasting periods) for each Experiment (Sklearn)

In this implementation the best predicted forecasting period was 1st year using Mean Imputation. This is the same best forecasting period achieved when using the first principles implementation. The difference being that it was achieved using Random Forest rather than Decision Tree. The plots for this forecasting period can be shown in Figure 3.5. When comparing the confusion matrix with the confusion matrix shown in Figure 3.4 there is a difference of +93.28 for True Positive, -95.6 for False Positive, -31.92 for False Negative, +32.92. Although this difference in performance shows that this best model is better than the best model of the first principal the differences are minor when using such metric. Comparing both ROC curves of the best forecasting period as described above there is a difference of +0.04 in AUC which again is very low. The ROC curves can be shown in Figure 3.5 (**sklearn**) and Figure 3.4 (first principles).

So to summarize the main differences between the third party library and the first principle implementation is the execution time and the best performing model which is Random Forest in **sklearn** and Decision Trees in first principle implementation. Altought the best model is different all accuracies and F1 scores are in general similiar to each other.



(a) Confusion Matrix



(b) ROC curve

Figure 3.5: Plots for best performing Model (Random Forest) using best imputation technique (Mean); Confusion Matrix using averaged values of all forecasting periods and ROC curve for best forecasting period (1st Year) using **sklearn**

Chapter 4: Conclusion

For the experiments done, the final metrics to evaluate their performance, are the averaged out accuracies and F1 scores of all forecasting periods as shown in Table 3.3. It was primarily noted that the Mean Imputation technique on missing data performed better than the Mode Imputation technique for the implementations from first principles of all models, and also for all the feature selection and dimensionality reduction techniques. The only model that performed nearly as good with Mode Imputation compared to Mean Imputation was Logistic Regression, with discrepancies of the averaged accuracy and F1 scores being;

- -0.31% and [-0.3%, +0.6%], respectively for the Oversampled dataset
- -0.62% and [-2.7%, +0.1%], respectively for the Oversampled PCA dataset
- -0.45% and [+0.8%, -3.5%], respectively for the Oversampled Chi2 datasets
- -0.02% and [0%, +0.001%], respectively for the Oversampled RFE dataset

The accuracy metric for all models when using Mean Imputation was between the range of +0.02% to +3.84% when compared to Mode Imputation, thus making it the best Imputation technique in terms of overall accuracy. In terms of F1 score it was performing better by the range of [0%, 0.1%] to [+3.9%, +3.1%]. Therefore the Mean Imputation technique is selected for the best performing Imputation technique as the only model that had better performance when using Mode Imputation was the Logistic Regression, but this model in general performed poorly with accuracy scores between the ranges of 49.48% and 58.22%, while low F1 scores in ranges of [37.1%, 29.4%] to [56.6%, 55.3%]. Regarding Feature Selection and Dimensionality Reduction techniques, RFE and Chi2 where the best performing in this case. PCA was falling behind with accuracy score discrepancies as large as approximately 5% and F1 scores discrepancies as large as approximately [5%, 5%]. For Decision Tree and Random Forest implemented from first principles; with no Feature Selection or Dimensionality Reduction, and for RFE and Chi2 feature selection techniques the accuracy and F1 scores where;

- 93.16% and [93%, 93.3%], respectively for Decision Tree with Oversampled dataset
- 91.96% and [91.8%, 92.1%], respectively for Decision Tree with Oversampled Chi2 dataset
- 92.06% and [91.9%, 92.2%], respectively for Decision Tree with Oversampled RFE dataset
- 88.11% and [87.8%, 88.4%], respectively for Random Forest with Oversampled dataset
- 89.01% and [89.2%, 88.8%], respectively for Random Forest with Oversampled Chi2 dataset
- 88.96% and [89.2%, 88.7%], respectively for Random Forest with Oversampled RFE dataset.

From the final results above, Decision Tree with Oversampled dataset with no feature selection and reduction, performed best with an accuracy score of 93.16% and F1 score of [93%, 93.3%], while the Random Forest with Oversampled dataset with no feature selection and reduction, performed best from the implementations done with **sklearn** learn with an accuracy score of 96.19% and F1 score of [96.2%, 96.2%].

For future work other imputation techniques which were not investigated in this project or the cited work [3] done on this problem, should be investigated. Also Feature Reduction and Feature Selection should be tested with a range of maximum features to be selected or reduced. Furthermore, other Feature Reduction techniques such as Linear Discriminant Analysis could be implemented and compared, and as for Feature Selection, methods such as Genetic Algorithms could be implemented and compared. Hyper parameters used in the models could also be investigated.

References

[1] M. Zikeba, S. K. Tomczak, and J. M. Tomczak, “Polish companies bankruptcy dataset,” http://archive.ics.uci.edu/ml/datasets/Polish+companies+bankruptcy+data, 2016, data retrieved from University of Science and Technology, Warsaw, Poland.

[2] “EMIS emerging markets information service,” https://www.emis.com/, accessed: 2019-01-03.

[3] S. K. M. Sai Surya Teja Maddikonda, “Bankruptcy prediction: Mining the polish bankruptcy data,” https://github.com/smaddikonda/Bankruptcy-Prediction, 2018, project Report for Polish Bankruptcy Prediction Dataset.

[4] ResidentMario *et al.*, “Missinggo: Open source missing data visualization module for Python,” 2016–, [Online; accessed 20/12/2018]. [Online]. Available: https://github.com/ResidentMario/missingno

[5] M. Waskom *et al.*, “Seaborn: Open source library for making statistical graphics in Python,” 2013–, [Online; accessed 20/12/2018]. [Online]. Available: https://pypi.org/project/seaborn/

[6] E. Jones, T. Oliphant, P. Peterson *et al.*, “SciPy: Open source scientific tools for Python,” 2001–, [Online; accessed 20/12/2018]. [Online]. Available: http://www.scipy.org/

[7] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, “Smote: synthetic minority over-sampling technique,” *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[8] L. Breiman, J. Friedman, R. Olshen, and C. Stone, “Classification and regression trees, 1984: Belmont,” *CA: Wadsworth International Group*, 1984.

[9] C. Gini, “Variabilità e mutabilità,” *Reprinted in Memorie di metodologica statistica (Ed. Pizetti E, Salvemini, T). Rome: Libreria Eredi Virgilio Veschi*, 1912.

[10] T. K. Ho, “Random decision forests,” in *Document analysis and recognition, 1995., proceedings of the third international conference on*, vol. 1. IEEE, 1995, pp. 278–282.

[11] R. D. McKelvey and W. Zavoina, “A statistical model for the analysis of ordinal level dependent variables,” *Journal of mathematical sociology*, vol. 4, no. 1, pp. 103–120, 1975.

[12] D. Timmerman, A. C. Testa, T. Bourne, E. Ferrazzi, L. Ameye, M. L. Konstantinovic, B. Van Calster, W. P. Collins, I. Vergote, S. Van Huffel *et al.*, “Logistic regression model to distinguish between the benign and malignant adnexal mass before surgery: a multicenter study by the international ovarian tumor analysis group,” *Journal of Clinical Oncology*, vol. 23, no. 34, pp. 8794–8801, 2005.

[13] M. Debella-Gilo and B. Etzelmüller, “Spatial prediction of soil classes using digital terrain analysis and multinomial logistic regression modeling integrated in gis: Examples from vestfold county, norway,” *Catena*, vol. 77, no. 1, pp. 8–18, 2009.

[14] G. E. Box and D. R. Cox, “An analysis of transformations,” *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 211–252, 1964.

[15] F. Mosteller and J. W. Tukey, “Data analysis. including statistics. handbook of social psychology.(g. lindzey and e. aronson, eds.) vol. 2, chapter 10, 80-203,” 1968.

[16] M. Kuhn and K. Johnson, *Applied predictive modeling*. Springer, 2013, vol. 26.

[17] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik, “Gene selection for cancer classification using support vector machines,” *Machine learning*, vol. 46, no. 1-3, pp. 389–422, 2002.

[18] X. Jin, A. Xu, R. Bie, and P. Guo, “Machine learning techniques and chi-square feature selection for cancer classification using sage gene expression profiles,” in *International Workshop on Data Mining for Biomedical Applications*. Springer, 2006, pp. 106–115.

[19] M. E. Tipping and C. M. Bishop, “Probabilistic principal component analysis,” *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 61, no. 3, pp. 611–622, 1999.

[20] ubershmekel, “arff: Open source library arff file type reader for Python,” 2012–, [Online; accessed 20/12/2018]. [Online]. Available: https://code.google.com/archive/p/arff/wikis/Documentation.wiki

[21] wesm *et al.*, “Pandas: Open source library providing high-performance, easy-to-use data structures and data analysis tools for for Python,” 2011–, [Online; accessed 20/12/2018]. [Online]. Available: https://pandas.pydata.org/

[22] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, “API design for machine learning software: experiences from the scikit-learn project,” in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

[23] glemaitre *et al.*, “Imbalanced-learn: Open source library to tackle the curse of imbalanced datasets Python,” 2016–, [Online; accessed 20/12/2018]. [Online]. Available: https://imbalanced-learn.readthedocs.io/en/stable/

[24] T. Oliphant, “NumPy: A guide to NumPy,” USA: Trelgol Publishing, 2006–, [Online; accessed <today>]. [Online]. Available: http://www.numpy.org/

[25] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Seljebotn, and K. Smith, “Cython: The best of both worlds,” *Computing in Science Engineering*, vol. 13, no. 2, pp. 31 –39, 2011.