Classifying credit score using different models and evaluating their performance

Ramsamy John Michael Geordan

September 2024

Table of Contents

List of Figures

List of Tables

Abstract

Background

Financial institutions, mostly banks, need to evaluate the likelihood of a potential borrower defaulting. Inaccurate predictions on the creditworthiness of a borrower will result in a loss of potential revenue or, in extreme cases, institution closure, which can spiral out into a financial crisis. Providing credit, leads to inherent credit risks that the financial institution will have to manage. The importance of accurate risk prediction is underscored by the 2007 US sub-prime mortgage crisis that lead to the 2007-2008 financial crisis.

Objective

The objective is to utilize prominent machine learning and traditional statistical models predict borrower default. The performance of these models on different datasets is then compared to find which model performs the best.

Methods

Seven common statistical and machine learning methods are used in this benchmarking study across four datasets commonly used in credit scoring research. Exploratory data analysis is performed on the datasets to identify any issues and drop redundant features. Multiple metrics are applied after testing the models using cross validation to rank the models across the four datasets. Because of class imbalance in the datasets, a stratified sampling method is used selecting training and testing samples.

Results and Conclusion

The results of our analysis have shown that the Random Forest classifier consistently outperformed the other models in all four datasets. It was able to handle class imbalance, different feature types and dataset sizes well while being easy to implement. The Support Vector Machine classifier came a close second while Decision Tree performed worse than the other models. Despite using stratified sampling when training our models, the higher the imbalance, the

worse our models were able to correctly classify defaults. The findings suggest that class imbalance remains a significant issue when applying our models. Hence, it is important to use proper sampling strategies to balance the training sets.

0.1 Datasets

The datasets used will be the German Credit (GC) dataset (Hofmann, 1994) with the personal status and sex feature changed to only sex, the Give Me Some Credit (GMSC) dataset from Kaggle (Credit Fusion, 2011), the Taiwan Credit (TC) dataset (I-Cheng Yeh, 2009), and the Australian Credit (AC) (Quinlan, 1987).

The GC dataset was collected by Prof. Hofmann and has 1000 observations with 20 features. In a 2011 competition, Kaggle introduced GMSC which they received from a financial instituion. It has 150000 observations and 12 features. The TC dataset has 25 features and 30000 observations. Finally, the AC dataset has 14 features with 690 observations. This dataset in particular had most of the customer identifying information removed, including variable names. We do however, have information on whether a variable is categorical or numeric.

List of Abbreviations

GC German Credit

GMSC Give Me Some Credit

TC Taiwan Credit

AC Australian Credit

NB Naive Bayes

KS Kolmogorov-Smirnov

TN True Negatives

TP True Positives

FN False Negatives

FP False Positives

TPR True Positive Rate

TNR True Negative Rate

FPR False Positive Rate

AUC Area Under Curve

ROC Receiver Operating Characteristic

DA Discriminant Analysis

LDA Linear Discriminant Analysis

KNN K nearest neighbours

EDA Exploratory Data Analysis

ANN Artificial Neural Network

MLP Multi-layer Perceptron

DT Decision Tree

RF Random Forest

SVM Support Vector Machine

LR Logistic Regression

RF Random Forest

NB Naive Bayes

CART Classification and Regression Trees

Chapter 1

Introduction

Credit scoring refers to the formal process of finding the likelihood of an applicant defaulting on their repayments (Hand and Henley, 1997). The objective to classify applicants into good credits (those that are least likely to default) and bad credit (those with the highest likelihood of default). There have traditionally been two techniques used to judge the creditworthiness of a borrower. The first one is a subjective method, based on the opinion of a credit analyst who makes decision based on experience and domain knowledge (Abdou and Pointon, 2011). The second one is a more modern method of using credit scores, and probability of default. These are calculated using past data and machine learning as well as statistical techniques. The issue with judgemental assessments is that the quality of decisions is solely based on the individual, who can have biases, and there is no consistency between different analysts' opinion. It is also more time consuming and analysts cannot review every possible applicant and need a way to pre-emptively discard unworthy applicants. All major banks have credit analysts that use credit scoring with specialised tools and software to score applicants, monitor their performance and manage their accounts.

While credit scoring primarily focuses on individual credit risk, household debt is a broader concern, as it represents a significant share of GDP in many major economies. Household debt saw a peak during the global financial crisis of 2008 with countries such as the United States and the United Kingdom reaching a peak of about 97% of GDP in 2009 while Germany was at 62%. India saw a huge spike during this period from 9% to 40% and has remained near this level. Following the crisis, countries like Mauritius and China has seen a consistent up trend in household debt as a percentage of GDP, underscoring broader shifts in borrowing behaviours. The Covid-19 pandemic has worsened this trend, which we can observe as another peak in household debt in 2020 (Data from IMF).

The increasing levels of household debt have placed pressure on banks to extend as much credit as possible. However, not all applicants can be granted credit, necessitating the use of robust methods to predict borrower default risk. Research in credit scoring, particularly through bench-marking studies, has explored the challenges and benefits of different approaches. While the

fundamental goal of credit scoring—assessing a borrower's likelihood of default—has remained constant, the techniques used have evolved significantly. As the performance of individual models reaches its limits, more advanced methods such as ensemble models and techniques for dimensionality reduction are being employed to further enhance accuracy. Improvements in predictive accuracy, driven by machine learning innovations, along with the scalability and ease of implementation of these models, will directly contribute to reducing financial losses for institutions.

The purpose of this study is to replicate existing benchmarks and extend the analysis of model performance using nested stratified cross-validation. We incorporate a diverse set of datasets, some of which are imbalanced, with a majority of non-default cases, while others are more balanced. This distribution and other relationships between the variables is examined in detail in the exploratory data analysis (EDA) section. Although the increasing complexity of black-box models in statistical learning poses challenges for interpretation, basic descriptive statistics can still help to derive insights from the characteristics of the datasets.

Chapter 2

Literature Review

2.1 Evolution of credit scoring

The field of credit scoring has been advancing consistently over the past several decades driven by a need for better methods to assess the creditworthiness of credit applications (Hand and Henley, 1997). This section will summarize the related works and literature that exists in the area, specifically those relating to the applications of machine learning and statistical models as well as benchmarking studies.

Credit scoring finds its roots in times when loan applications were decided purely based on experience and human intuition. Judgemental techniques was the only way to assess the creditworthiness of an applicant. J. Crook, 1996 was one of the pioneers who did a review of credit scoring techniques at the time and divided them into judgemental and credit scoring methods. He highlighted the shortcomings of judgemental techniques which are prone to bias and emphasized how credit scoring techniques could add consistency and reliability to the approval decisions.

Some of the more popular models which will be referred to as the classical models in credit scoring include Logistic Regression (LR), Discriminant Analysis (DA), Decision Tree (DT), Artificial Neural Network (ANN), Naive Bayes (NB) and Random Forest (RF). LR has been an industry standard due to its simplicity and respectable performance (J. N. Crook, Edelman, and Thomas, 2007). Despite the existence of so many possible models, due to the dynamic nature of technology and the different environments in which they are applied, no model has been crowned as the best performing one (Abdou and Pointon, 2011). Most studies that involved credit scoring use either some or all of these models.

Although most studies will try to classify applicants into good and bad credit, Yeh and Lien, 2009 argues that the prediction of a probability of default is more useful from the point of view of risk control compared to only classifying clients into binary results. He found that ANN was the only one that gave reliable predicted probability of default from the models. Hence, he says that ANN should be used more than logistic regression.

As the field evolved, researchers have tried to compare the performance of different models on a larger scale. Benchmarking studies have shaped the direction of model applications by identifying the strengths and weaknesses of the different models. Baesens et al., 2003 was the first to do a benchmarking study with multiple classifiers and metrics across different datasets. The study was ground-breaking in its scope and became a reference for credit scoring. Later, Lessmann et al., 2015 did an update of this paper to keep up with the changes in machine learning techniques. Lessmann et al., 2015 used 41 classifiers on 8 credit scoring datasets and evaluated their performance using 6 metrics. They concluded that heterogeneous ensemble classifiers performed better than traditional classifiers like LR. They also found that multiple metrics gave a similar assessment of the performance of the classifiers, and recommended that two additional metrics be used like the brier score to assess the accuracy of scorecards and the H measure to cover areas where the AUC doesn't do well. Their findings have also encouraged researches to use ensemble models in credit scoring. These studies could be used as a yardstick to which future research could compare themselves to without having to test each model themselves.

Class imbalance is a common issue in credit scoring datasets. Hence, it is important to find ways to reduce the effects of these imbalances when modelling and deploying our models. Class imbalance can artificially inflate accuracy scores for example and lead to being too optimistic about how our model performs. Brown and Mues, 2012 did a study on how class imbalance could affect the performance of machine learning models by under-sampling minority classes of 5 real word datasets. He found that random forest and gradient boosting classifiers performed well in this context.

Stelzer, 2019 did a similar study, comparing how well machine learning and statistical methods perform under class imbalance but her paper tried to reduce the effects of a small minority class by using sampling techniques. Sampling strategies used to mitigate class imbalance include up sampling, down sampling, Synthetic Minority Oversampling Technique (SMOTE), and Random Over Sampling Examples approach (ROSE). She concluded that using simple sampling strategies gave better results than more complicated ones and that using sampling techniques improved the performance of the best performing classifiers. She also found that homogeneous classifiers like RF performed well with and without sampling.

Rather than trying to find the best performing model, some studies have instead tried to focus on how to improve the results of the already known models like decision trees. Hybrid machine learning can be a solution to this problem by combining machine learning with other algorithms to improve them. Weng and Huang, 2021 found that feature selection using expectation maximization and instance selection with information gain formula, together with decision trees also lead to improvements in the performance of the classifier instead of using either one or none. Weng and Huang, 2021 also found that this model, called Clustering Based Decision Trees (CBDT), outperformed the classical classifiers. Other classifiers also saw an increase in performance based on the metrics applied in the study (F1, Accuracy, and CostEffect), compared to simply applying the base model.

With advancements in artificial intelligence and deep learning, there has been increasing interest in applying more complex learning models to credit scoring. Addy et al., 2024 found that convolutional Neural Networks (CNN) and especially Deep Sequential Neural Network (DSNN) are very well performing and often better performing than classical models. However, the more complex the model, the harder it becomes to interpret and understand which variables are the biggest contributors to the predictions and why. It is also more difficult to deploy as it requires more fine tuning and time to train the models. In Addy et al., 2024 he also didn't compare the deep learning models with random forest or use many different metrics.

2.2 Previous findings

In Baesens et al., 2003, they showed that ensemble classifiers ranked better than individual models. Figures 2.1 and 2.2 illustrate the rankings of all classifiers used in the study. In their study, ANN had the best rank for individual models, while random forests had the best rank for homogeneous ensembles as well as overall. Meanwhile HCES-Bag had the best rank for heterogeneous ensembles.

TABLE 7: AVERAGE RANKS OF INDIVIDUAL CLASSIFIERS AND HOMOGENEOUS ENSEMBLES

		AvgR-AUC		AvgR-PCC		AvgR-BS		AvgR-H-Measure		AvgR
Individual classifiers	ANN	5.95	(.6505)	7.93	(.1008)	14.17	**(.0000)**	6.88	**(.0377)**	8.73
	B-Net	14.25	**(.0000)**	13.49	**(.0000)**	8.90	**(.0000)**	14.38	**(.0000)**	12.75
	CART	21.52	**(.0000)**	19.18	**(.0000)**	21.07	**(.0000)**	21.01	**(.0000)**	20.70
	ELM	16.91	**(.0000)**	16.93	**(.0000)**	22.65	**(.0000)**	16.75	**(.0000)**	18.31
	ELM-K	8.49	**(.0009)**	8.83	**(.0108)**	22.57	**(.0000)**	8.16	**(.0008)**	12.01
	J4.8	20.17	**(.0000)**	17.30	**(.0000)**	10.67	**(.0000)**	19.16	**(.0000)**	16.82
	k-NN	16.43	**(.0000)**	17.22	**(.0000)**	14.84	**(.0000)**	17.04	**(.0000)**	16.38
	LDA	12.05	**(.0000)**	10.85	**(.0000)**	7.23	**(.0004)**	11.56	**(.0000)**	10.42
	LR	11.03	**(.0000)**	10.46	**(.0000)**	5.17	(.1556)	10.69	**(.0000)**	9.34
	LR-R	11.44	**(.0000)**	11.07	**(.0000)**	18.44	**(.0000)**	11.49	**(.0000)**	13.11
	NB	18.06	**(.0000)**	17.99	**(.0000)**	13.30	**(.0000)**	18.34	**(.0000)**	16.92
	RbfNN	18.03	**(.0000)**	18.35	**(.0000)**	15.68	**(.0000)**	18.51	**(.0000)**	17.64
	QDA	17.57	**(.0000)**	16.96	**(.0000)**	9.34	**(.0000)**	17.72	**(.0000)**	15.40
	SVM-L	11.51	**(.0000)**	12.00	**(.0000)**	17.54	**(.0000)**	11.99	**(.0000)**	13.26
	SVM-Rbf	9.68	**(.0000)**	10.46	**(.0000)**	18.22	**(.0000)**	9.98	**(.0000)**	12.08
	VP	23.23	**(.0000)**	23.15	**(.0000)**	19.94	**(.0000)**	23.28	**(.0000)**	22.40
Homogeneous ensemble classifiers	ADT	9.45	**(.0000)**	8.58	**(.0203)**	4.66	(.3003)	9.01	**(.0000)**	7.92
	Bag	4.85	(.9471)	6.07	(.6868)	7.21	**(.0004)**	4.43	(.9025)	5.64
	BagNN	4.76	(.9471)	6.44	(.6868)	3.57	(.7019)	5.26	(.7007)	5.01
	Boost	6.10	(.6298)	7.40	(.2678)	13.01	**(.0000)**	6.85	**(.0377)**	8.34
	LMT	12.48	**(.0000)**	12.48	**(.0000)**	6.11	**(.0153)**	13.08	**(.0000)**	11.04
	RF	**4.69**	/	**5.66**	/	**3.18**	/	**4.31**	/	**4.46**
	RotFor	11.35	**(.0000)**	10.94	**(.0000)**	13.28	**(.0000)**	10.07	**(.0000)**	11.41
	SGB	10.00	**(.0000)**	10.23	**(.0001)**	9.25	**(.0000)**	10.05	**(.0000)**	9.88
Friedman χ^2_{23}		1323.61	**(.0000)**	1026.72	**(.0000)**	1654.40	**(.0000)**	1298.26	**(.0000)**	

Values in brackets represent the adjusted *p*-value corresponding to a pairwise comparison of the row classifier to RF. Bold face indicates significance at the 5%, an underscore significance at the 1% level. To account for the overall number of pairwise comparisons, we adjust *p*-values using the *Rom*-procedure (García et al., 2010). Prior to conducting multiple comparisons, we employ the Friedman test to verify that at least two classifiers

Figure 2.1: Baesens et al ranking of individual and homogeneous classifiers

TABLE 8: AVERAGE RANKS OF HETEROGENEOUS ENSEMBLE CLASSIFIERS

		AvgR-AUC		AvgR-PCC		AvgR-BS		AvgR-H-measure		AvgR
No ensemble selection	AvgS	7.21	**(.0004)**	7.59	**(.0333)**	5.90	**(.0155)**	7.14	(.0807)	6.96
	AvgW	6.05	**(.0152)**	8.16	**(.0052)**	7.20	**(.0001)**	5.61	(.6313)	6.76
Static direct ensemble selection	Top-T	6.16	**(.0152)**	8.75	**(.0003)**	8.27	**(.0000)**	6.66	(.1980)	7.46
	BBM	10.04	**(.0000)**	9.58	**(.0000)**	9.13	**(.0000)**	9.99	**(.0000)**	9.68
	CompM	11.77	**(.0000)**	9.49	**(.0000)**	17.10	**(.0000)**	11.20	**(.0000)**	12.39
	EPVRL	6.77	**(.0028)**	7.69	**(.0275)**	6.01	**(.0145)**	7.41	**(.0349)**	6.97
	GASEN	7.35	**(.0002)**	7.54	**(.0333)**	6.11	**(.0120)**	6.88	(.1635)	6.97
	HCES	6.35	**(.0149)**	7.27	(.0551)	5.54	**(.0271)**	6.73	(.1980)	6.47
	HCES-Bag	**4.14**	/	**5.52**	/	**3.83**	/	**5.24**	/	**4.68**
	MPOCE	7.46	**(.0001)**	7.22	(.0551)	7.59	**(.0000)**	7.40	**(.0349)**	7.42
	Stack	14.80	**(.0000)**	13.58	**(.0000)**	16.96	**(.0000)**	14.58	**(.0000)**	14.98
Static indirect ensemble selection	CuCE	7.45	**(.0001)**	7.71	**(.0275)**	8.27	**(.0000)**	7.84	**(.0071)**	7.82
	k-Means	8.59	**(.0000)**	8.06	**(.0076)**	7.87	**(.0000)**	7.59	**(.0197)**	8.03
	KaPru	15.77	**(.0000)**	14.72	**(.0000)**	11.20	**(.0000)**	15.90	**(.0000)**	14.40
	MDM	12.40	**(.0000)**	11.17	**(.0000)**	8.34	**(.0000)**	11.72	**(.0000)**	10.91
	UWA	6.00	**(.0152)**	6.89	(.0755)	11.57	**(.0000)**	6.46	(.2275)	7.73
Dynamic ensemble selection	KNORA	14.79	**(.0000)**	14.08	**(.0000)**	15.19	**(.0000)**	15.00	**(.0000)**	14.76
	PMCC	17.91	**(.0000)**	15.97	**(.0000)**	14.91	**(.0000)**	17.66	**(.0000)**	16.61
Friedman χ^2_{23}		945.1	**(.0000)**	559.0	**(.0000)**	953.5	**(.0000)**	867.3	**(.0000)**	

Figure 2.2: Baesens et al ranking of individual and heterogeneous classifiers

From J. N. Crook, Edelman, and Thomas, 2007, we get the following table that summarizes the accuracy scores of some models on the German Credit and Australian Credit dataset.

	German Credit	Australian Credit
Superior models	MOE	MOE
	RBF	RBF
	MLP	MLP
	Logistic regression	Logistic regression
		LDA
		K-NN
Inferior models	LVQ	LVQ
	FAR	FAR
	LDA	Kernel density
	K-NN	CART
	Kernel density	
	CART	

Table 2.1: Comparison of Superior and Inferior Models

These set a precedent to which we can add to after doing our own comparisons.

Chapter 3

Methodology

This section describes the procedure for classifying good credit and the classification models used in this study.

If you have a set of features for each applicant, x_i, where i is the number of customers ranging from 1 to n, we can model defaults with a variable y_i which is 0 if they are a good customer or 1 if they defaulted on their loan. Using x_i, we can estimate the probability of y_i being 1 or 0. By defining $p(y_i = 0|x_i) = p(+|x_i)$ as the probability that customer i won't default and $p(y_i = 1|x_i) = p(-|x_i)$ as the probability that they will, we can compare each probability with a threshold τ. If $p(+|x_i) < \tau$ then we can say that it was a bad credit and vice versa (Stelzer, 2019).

In this study, we use seven approaches to estimate $p(+|x_i)$ consisting of commonly used statistical and machine learning methods. These statistical models include logistic regression, discriminant analysis, k-nearest neighbours along with machine learning techniques such as classification trees, support vector machines, random forest, and artificial neural networks.

3.1 Pre-Processing

To ensure our models produce reliable results, it's essential to begin with clean, well-prepared data. The phrase "garbage in, garbage out" often employed in the field of computer science emphasizes that flawed data leads to flawed model outputs, making data pre-processing a crucial step in the data science work flow.

We start by encoding categorical variables, as they cannot be directly used by most models. This is done by converting them into binary variables through one-hot encoding. Next, we standardize features that follow a bell-shaped distribution, ensuring they have a mean of 0 and a variance of 1. This prevents features with wider ranges from having a disproportionate effect on the model. For features that are not normally distributed, we apply a robust scaler (*RobustScaler* 2024), which is less sensitive to outliers since it uses the interquartile range for scaling.

Missing values are addressed either through imputation of simple statistics like the median. However, outliers are not removed from the datasets, particularly because in cases such as the AC dataset, where the nature of the variables is unclear, and outliers may represent valid observations within the data distribution. Lessmann et al., 2015 also mentions that while outliers can negatively impact the performance of the individual models, it leaves the relative performance of each model unchanged. It is also interesting to notice the effect of outliers on each model and how well the latter perform despite the existence of the former. In real world settings, outliers are bound to be present and the best model should perform well in any situation.

3.2 Models

3.2.1 Logistic regression

The logistic regression is a regression model for classification that is used to predict an outcome variable using some predictors and a sigmoid function, $\sigma(z)$. The sigmoid function maps real valued numbers to a range (0,1). By making the two cases $P(y_i = 1|x_i)$ and $P(y_i = 0|x_i)$ sum to 1, we can get probabilities using the sigmoid function (Jurafsky and Martin, 2024).

$$P(y_i = 1|x_i) = \sigma(w \cdot x + b) = \frac{1}{1 + exp(-(w_i \cdot x_i + b))}$$
$$P(y_i = 0|x_i) = 1 - \sigma(w \cdot x + b) = \frac{1}{1 + exp(w_i \cdot x_i + b)}$$

We can make decisions by setting a decision boundary, τ.

$$\text{decision(x)} = \begin{cases} 1 \text{ if } P(y = 1|x) > \tau \\ 0 \text{ otherwise} \end{cases}$$

3.2.2 Discriminant analysis

First introuced by R.A Fisher in 1936, DA is a common statistical method that can be used in our classification task. DA makes use of bayes theorem that

$$P(Y = k \mid X = x) = \frac{P(X = x \mid Y = k)P(Y = k)}{P(X = x)}$$

so as to classify a given x_i into a class k based on the Mahalanobis distance. If the mean of x is close to the mean of the class k, μ_k, while the variance of the group is at its minimum, x will be classified into k.

The Mahalanobis distance of x from a probability distribution Q is found using:

$$d_M(\vec{x}, \vec{y}; Q) = \sqrt{(\vec{x} - \vec{y})^\mathsf{T} S^{-1} (\vec{x} - \vec{y})} \tag{3.1}$$

The probability density functions of the data for Linear Discriminant Analysis (LDA) and quadratic discriminant analysis, is assumed to be a multivariate Gaussian distribution with density[1]:

$$P(x|y=k) = \frac{1}{(2\pi)^{d/2}|\Sigma_k|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_k)^t \Sigma_k^{-1}(x-\mu_k)\right). \tag{3.2}$$

where d is the number of features. Our aim is to maximize $P(X = x \mid Y = k$ and using discriminant functions for x w.r.t k, $\delta_k(x)$ get a final decision rule where we predict that x belongs to class k if the discriminant function is largest.[2]

$$\hat{\delta}_k(x) = \log \hat{\pi}_k - \frac{1}{2}\hat{\mu}_k^T \hat{\mathbf{\Sigma}}^{-1} \hat{\mu}_k + x^T \hat{\mathbf{\Sigma}}^{-1} \hat{\mu}_k \tag{3.3}$$

where $\hat{\pi}_k$ is the fraction of samples in our training data that belongs to the class k. This decision boundary however, only applies when the covariance matrices Σ_k of each class k is the same. Otherwise, quadratic discriminant analysis is needed.

3.2.3 Support Vector Machines

Support Vector Machine (SVM) introduced by Cortes and Vapnik, 1995 is a classifier that groups data using hyperplanes in high or infinite dimensional space. Support vector classifiers and the separating hyperplane with the largest margin between groups. The larger the margin, the better generalization of the classifier.

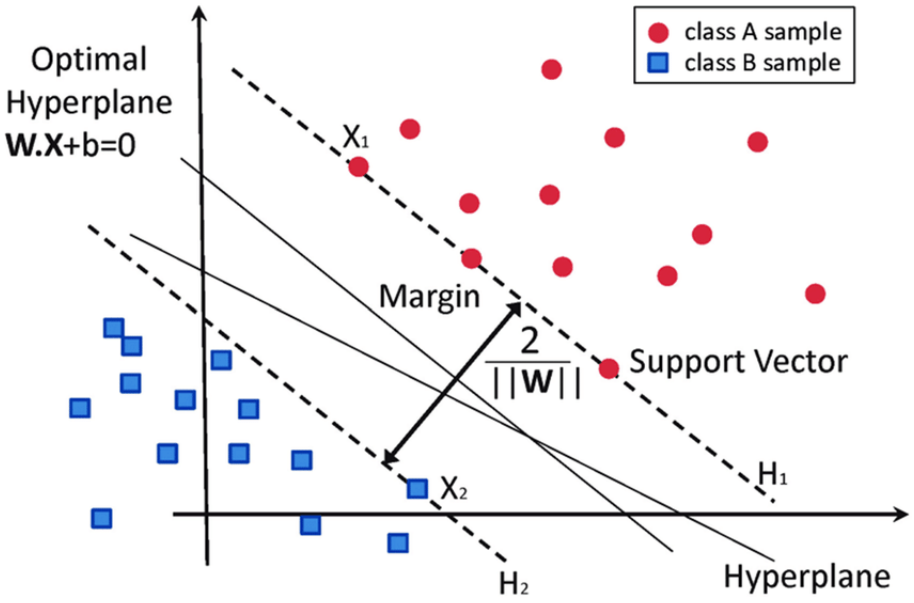


Figure 3.1: svm with hyperplane drawn
Source: Al Jallad, Desouki, and Aljnidi, 2020

[1] Hastie, Tibshirani, and Friedman, 2009.
[2] Bacallado and Taylor, 2024.

Figure 3.1 shows a hyperplane drawn in the middle of the 2 support vectors so as to classify observations as class a or class b. The optimal hyperplane is found at $w \cdot x + b = 0$, where w is the weight vector, x is the input vector, and b is the bias term. Depending on the problem to solve, the margin can either be a soft margin, which allows for misclassification errors or a hard margin expressed as $y_i(wx_i + b) \geq a$ (*What Is Support Vector Machine?* 2023).

For linearly separable data, the SVM kernel is linear. However, when the data is not linearly separable, SVMs can implement non-linear class boundaries using kernel functions, mapping input vectors into higher-dimensional feature spaces to achieve separability. In such cases, a radial basis function kernel is commonly used to add dimensions, making the data more easily separable by a hyperplane.

The SVM classifier is very slow when applied to larger datasets. *1.4. Support Vector Machines* 2024 mentions in the complexity section that the algorithm tries to solve a quadratic problem when non-linear kernels are applied. The complexity is $O(n_samples^2 * n_features)$. However, this is not an issue with linear kernels which uses the liblinear algorithm and hence scales well.

3.2.4 KNN

The K nearest neighbours (KNN) classifier is another statistical technique which is popular for credit scoring. KNN looks for similarity between and within groups based on euclidean distance. It groups observations that minimizes the mean distance within groups and tries to maximize the distance between means of different groups. The k that optimizes the means is chosen.

Figure 3.2: KNN clustering

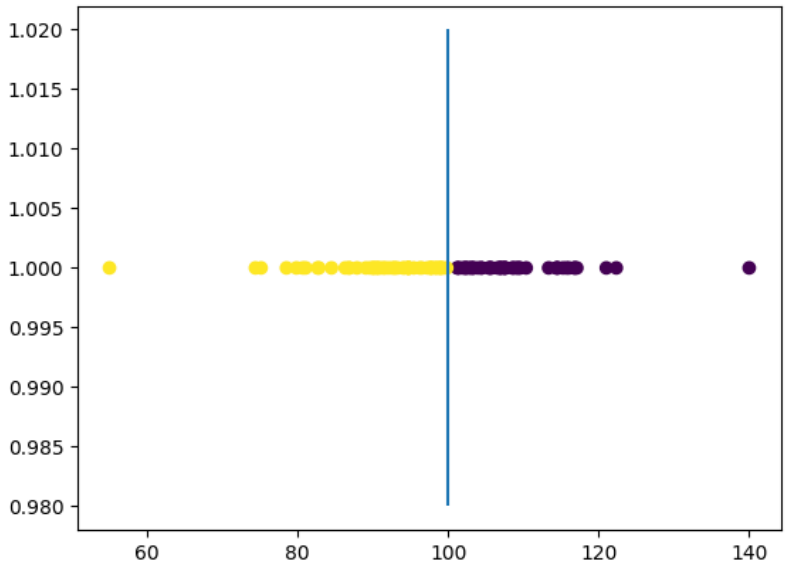


Figure 3.2 shows 2 groups separated with a k of 5. A lower k usually leads to a worse classification, especially if the data is messy. The value of k chosen will depend on the dataset and

setting a $k = 1$ means that only the single nearest observation will dictate the class of test data.

KNN has the advantage of being quick, easy and versatile to apply. However, it suffers with slowness when there is a large number of predictors. In case the data is noisy or not well separated, KNN also suffers from lower performance. KNN relies on good feature selection to get the best performance.

3.2.5 Tree Based Classifiers

Tree based classifier methods considered in this paper are Classification and Regression Trees (CART) and random forest which is an ensemble method of CART.

Decision trees

CART is an algorithm for building decision trees but other methods include ID3, C4.5 and C5.0 which were developed by Quinlan. CART partition the dataset into rectangular regions and assigns a constant value c to each partition. Since in credit scoring our target is binary, we will make use of classification trees. The process begins by splitting each variable into two recursively, thereby creating a tree structure where each node represents a decision point, until the final decision called a terminal node.

The splits are decided using the impurity measure $Q_m(T)$ which is usually calculated using either the gini index or the cross-entropy. For each node m, representing a region R_m with N_m observations, the proportion of observations belonging to a class k, $\hat{p}_{mk}$ is given by[3]:

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k)$$

We then say that the observations in node m belong to class k if $\hat{p}_{mk}$ is the maximum between all classes.

The gini is calculated using:

$$\sum_k p_{mk}(1 - p_{mk})$$

and cross entropy using:

$$-\sum_k p_{mk} \log(p_{mk})$$

Once a stopping criteria is met, like a maximum tree size or a minimum node size, the tree stops growing. This full sized tree can then be pruned using a cost complexity criterion so as to prevent over-fitting. The problem with CART is that it has a high variance and a small change

[3]Hastie, Tibshirani, and Friedman, 2009.

in the data can lead to a completely different splits which can make interpretation of the tree unreliable[4]. Bagging methods like random forest will alleviate this problem.

Random forest

RF, developed by Breiman, 2001, combine tree models and improve predictions by incorporating the principles of bagging (bootstrap aggregating). RF builds multiple decision trees using bootstrap samples of the training data, where each tree is generated from a randomly selected feature at each node, rather than considering all features. This randomization introduces additional diversity among the trees, reducing the correlation between them and enhancing the robustness of the final model (Hastie, Tibshirani, and Friedman, 2009).

After constructing a large ensemble of trees, the final prediction for a given sample is determined by a majority vote of all the trees' predictions for classification tasks, or by averaging their outputs for regression tasks. Thus, if $\hat{C}_b(x)$ represents the class prediction of the bth tree, the overall classification rule for Random Forests is (Hastie, Tibshirani, and Friedman, 2009):

$$\hat{C}_{rf}^{B}(x) = \text{majority vote } \{\hat{C}_b(x)\}_1^B \tag{3.4}$$

Studies in credit scoring have shown the effectiveness of RF, with Lessmann et al., 2015 and Brown and Mues, 2012 recommending its use in this context.

1.11. Ensembles 2024 implements RF classifiers using 2 sources of randomness. First by randomly selecting bootstrapped samples from the training set and also a split using a random subset of features. This randomness aims to decrease the variance of the RF estimator. As mentioned earlier, individual decision trees are prone to a high variance and often overfit the data. Hence, the randomness will lead to a lower correlation between the prediction errors of the trees. By taking an average of those predictions, some errors can cancel out. The sklearn implementation however, in contrary to Breiman, 2001 combines classifiers by averaging their probabilistic prediction, instead of letting each classifier vote for a single class.

3.2.6 ANN

ANN are a class of models inspired by the structure and functioning of the human brain's neurons. They consist of an input layer (input nodes), one or more hidden layers, and an output layer (output nodes). Each node (or neuron) is associated with an activation function, which determines whether the neuron should be activated based on the input it receives. The connections between the neurons in the layers allow the network to learn and model complex patterns in data.

[4]Hastie, Tibshirani, and Friedman, 2009.

3.2.7 MLP

A Multi-layer Perceptron (MLP) is a type of feed-forward ANN which has at least one hidden layer. MLPs can be used for classification and are able to capture complex patterns in our dataset.

Figure 3.3 shows a one hidden layer MLP. The layer on the left, the input layer, contains the input variables such as age, marital status and loan duration ($x_i|x_1, x_2, ..., x_m$). Each neuron in the hidden layer will have a weighted sum of each variable and an activation function applied on that weighted sum. Finally, the output layer will receive these values from the last hidden layer and transform them into outputs (*1.17. Neural Network Models (Supervised)* 2024).

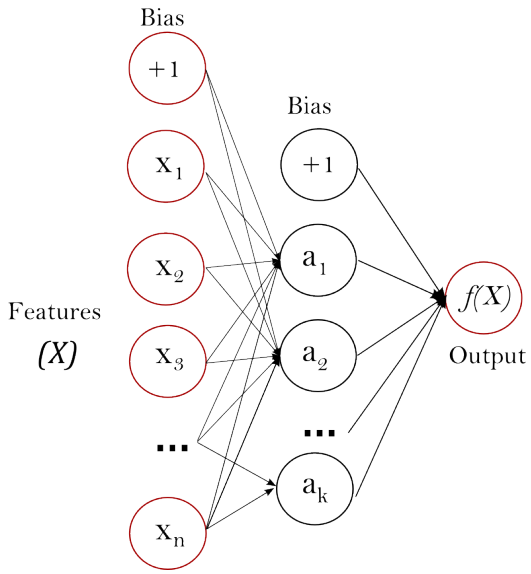


Figure 3.3: MLP
Source: *1.17. Neural Network Models (Supervised)* 2024

There are many different activation functions which can be used. Rectified Layer Unit (ReLU) is a popular choice due to its simplicity. It is simply $ReLU(x) = max(x, 0)$. It outputs 0 for negative values and remains unchanged for positive values. Other popular activation functions include the hyperboloic tangent, tanh, and the sigmoid function (Dubey, Singh, and Chaudhuri, 2022).

$$\text{Tanh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.5}$$

$$\text{Sigmoid(x)} = \frac{1}{1 + e^{-x}} \tag{3.6}$$

The sigmoid function will return values between 0 and 1 while the tanh function will squish the values between -1 and 1. Each of these have their own advantages and limitations. For instance, the sigmoid function can lead to poor convergence of the model as it is saturated towards 0 and 1 for very low and very high input values.

The weights of the hidden layers are calculated using gradient descent, with the backpropagation algorithm employed to reduce the error between the predicted and actual outputs.

3.3 Performance Metrics

After having fitted the models, we will evaluate each of their performance with test samples by comparing the following metrics:

1. Accuracy
2. Precision
3. Recall/Sensitivity/True Positive Rate (TPR)
4. Specificity/True Negative Rate (TNR)
5. Cohen's κ
6. AUC and ROC curve
7. Kolmogorov-Smirnov (KS)

3.4 The confusion matrix

The values needed to calculate those metrics can be used to construct a confusion matrix. It shows True Positives (TP), i.e. how many outcomes were correctly classified as positive, True Negatives (TN), i.e. how many outcomes were correctly classified as negative, False Positives (FP), i.e. how many outcomes were incorrectly classified as positive and finally False Negatives (FN), i.e. how many outcomes were incorrectly classified as negative.

Figure 3.4: Confusion Matrix

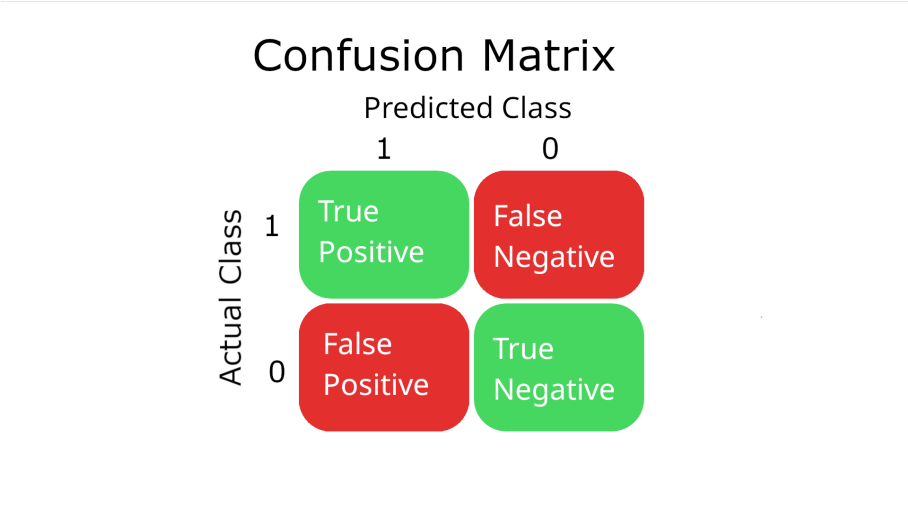


Figure 3.4 shows how a 2x2 confusion matrix looks like.

3.5 Accuracy, precision, recall and specificity

3.5.1 Accuracy

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FN + FP} \tag{3.7}$$

Accuracy tells us how often the model made a correct prediction overall. However, in the case where datasets are unbalanced, with a minority of defaults, just predicting positive or negative all the time gives a good accuracy score. So, the other metrics will be more useful.

3.5.2 Precision

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3.8}$$

Precision measures how often the model is correct when predicting the target class.

3.5.3 Recall/Sensitivity/True Positive Rate

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3.9}$$

Recall tells us whether the model can find all objects of the target class. A high TPR indicates a low Type II error rate.

Specificity/True Negative Rate

Specificity or TNR is a measure of how often the model correctly identifies true negatives. A higher specificity means a lower Type I error rate.

$$\text{Specificity} = \frac{TN}{TN + FP} \tag{3.10}$$

3.6 Cohen's κ

κ is generally thought to be a more robust measure than accuracy, as it takes into account the possibility of a high accuracy score occurring by chance.

$$\kappa = \frac{p_o - p_e}{1 - p_e} \tag{3.11}$$

where p_o is the probability of agreement (accuracy 3.7) and p_e is the probability of agreement by random chance.

$$p_e = \frac{TP + FN}{N} \cdot \frac{TP + FP}{N} + \frac{TN + FP}{N} \cdot \frac{FN + TN}{N}$$

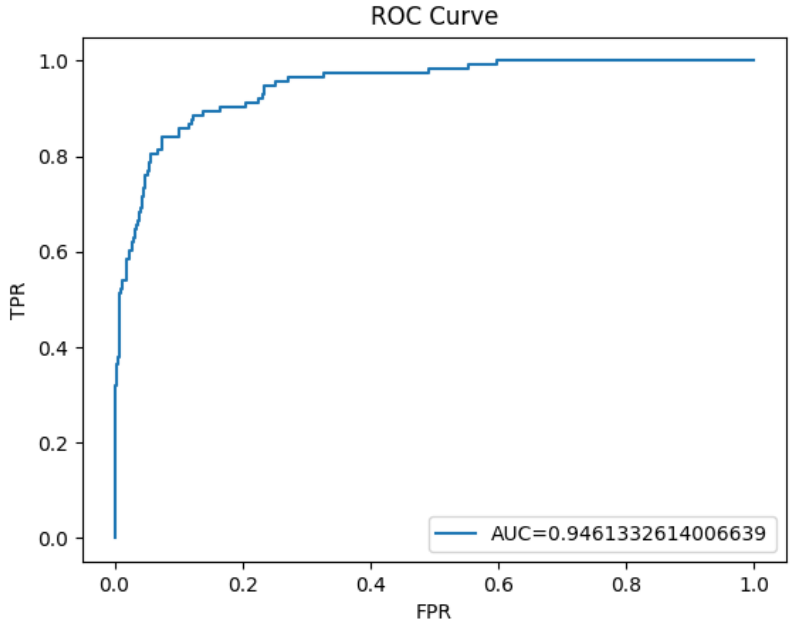
3.7 AUC and ROC curve

An ROC curve plots the TPR against the False Positive Rate (FPR) at each threshold setting. Meanwhile, FPR is defined as:

$$FPR = \frac{FP}{FP + TN} = 1 - TNR \tag{3.12}$$

and tells us how often the model incorrectly identified negatives as positive, i.e. the rate of making a Type I error.

Figure 3.5: ROC curve



The AUC of a classifier is equivalent to the probability that the classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance (Fawcett, 2006). A model that randomly guesses whether to classify an instance as positive rather than negative will have an AUC of 0.5. The maximum of the AUC is 1, which is a perfect model, indicating that it correctly predicted all positive instances.

3.8 Kolmogorov-Smirnov

A KS test is usually done to see how close the Cumulative Distribution Functions (CDFs) of two distributions are close by measuring the distance between the two CDFs. It can also be used to calculate how dissimilar two distributions are and by applying it on TPR and FPR, if they are very dissimilar then the model is performing well. An advantage of the KS test is that it ranges from 0 to 1 as compared to AUC score which ranges from 0.5 to 1 so, it is more intuitive to interpret. It is also not affected much by class imbalance.

$$\text{KS} = max(\text{TPR} - \text{FPR}) \tag{3.13}$$

3.9 Model Selection and parameter tuning

The purpose of this study is to compare different classifiers, and in doing so we also need to optimize model performance through proper hyper-parameter tuning. Hence, we need a way to achieve both of these goals while minimizing bias and also making the best use of our datasets. Two main approaches can be used at this end:

1. Repeated random splitting
2. Nested Cross Validation

Repeated sampling does multiple train-test splits at a fixed ratio. Each split is independent and by averaging our model performance over all these splits, we obtain an overall estimate of our model performance. However, this method has potential drawbacks, such as underutilizing parts of the dataset and the possibility of obtaining highly similar splits, which may limit the diversity of training and testing samples.

Nested cross validation does not have these issues. Although Wainer and Cawley, 2021 shows that this approach doesn't give much additional information relative to the computing cost of all these repeated cross validations, especially when comparing SVM, RF, NB and gradient boosting machines, nested cross validation still produces the most robust performance estimate we can get. It is also to be noted that MLP needs some hyper-parameter tuning which in principle should not be found using the whole dataset to avoid over-fitting.

There are two ways to test hyper-parameter combinations:

1. GridSearchCV

   This method tests every combination of parameters provided in a parameter grid. While it is the most extensive way to look for hyperparameters, it suffers from a limitation that it is very slow to test all combinations. Adding in cross validation, the number of models to train increases exponentially.

2. RandomizedSearchCV

   This method tests a fixed number of parameters randomly chosen from the parameter grid. Although it is not as extensive as a gridsearch, it still gives good approximations. As the number of parameters tested remains constant, it is independent of the number of parameters in the grid.

As some of these models take significant to time run multiple times. To save time on model tuning, a randomized search is done instead of a full grid search for some of them. It is also not feasible to test every possible parameter combination so each model will only perform at a

point close to their best but not necessarily at the optimum point. Another limitation of hyper-parameter tuning is that deciding what to include in the parameter grid is an exercise of heuristic so depending on the statistician's prior knowledge of the dataset, the parameters will change.

Something else to note is that the metric targeted for the model tuning is the f1 score. The f1 score combines precision and recall into a single metric so that we can reach a balance between the two measures when tuning our models.

3.9.1 Cross Validation and the stratified k-fold

Standard k-fold cross validation divides a dataset into k random subsets called folds and trains the model on $k - 1$ folds. The remaining fold is then used to validate the model. This process is repeated k times until all the folds have been trained and validated. An average of the model performance on each fold then gives information of what the best parameters for a model are as well as helps remove bias from tuning a model only for one training set and validating the model for only one testing set. Cross validation also removes bias by repeating the training process multiple times on different parts of the dataset.

Figure 3.6 illustrates a traditional cross validation work-flow goes by splitting a dataset into a training and testing set, performing cross validation to fine tune parameters and then doing a final evaluation on the testing set.

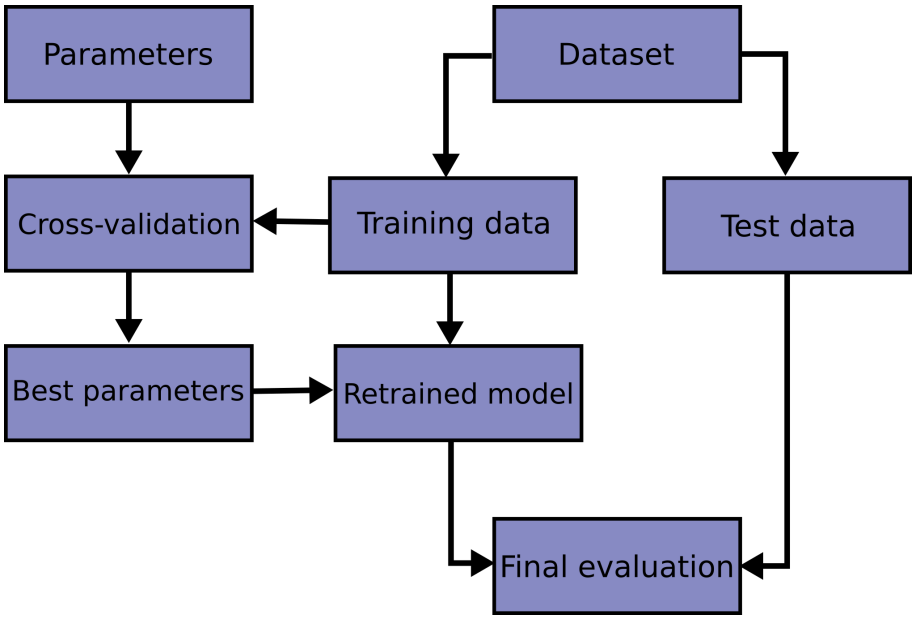


Figure 3.6: Cross validation work-flow.
Source: *3.1. Cross-validation* 2024

Given the class imbalance present in some of our datasets, an additional step can be incorporated to improve the accuracy of cross-validation results.

As noted by Rice, 1988, stratified random sampling involves partitioning the population into subgroups, or strata, which are then sampled independently. The results from each stratum are

combined to estimate population parameters, such as the mean. Stratified sampling generally yields more accurate estimates of population parameters with reduced variance compared to simple random sampling.

By applying this technique to our cross-validation process, we ensure that both the minority and majority classes are proportionally represented in each fold, preserving the class distribution of the entire dataset. This adjustment allows the models to be trained and validated on samples that reflect the true class proportions, leading to more reliable performance estimates.

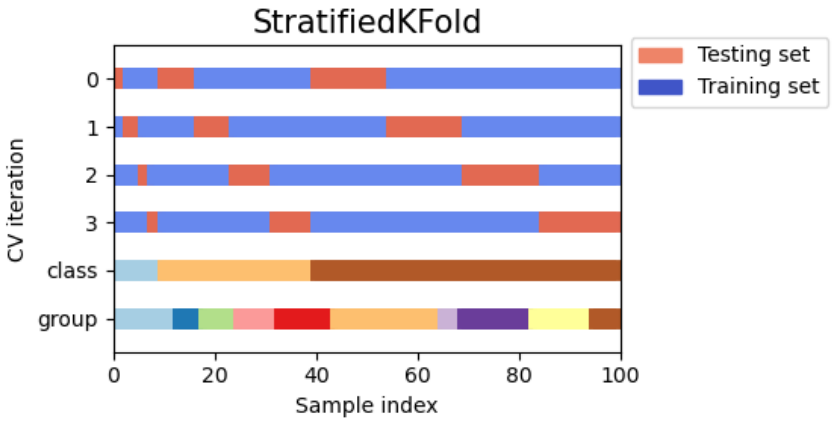


Figure 3.7: skfold
Source: *3.1. Cross-validation* 2024

Figure 3.7 illustrates how class representations are maintained in the cross validation samples.

In this paper a 5 fold stratified cross validation is applied for parameter tuning and the models are then tested with a different 10 fold cross validation. This ensures that our estimates are not too optimistic due to information leakage.

Chapter 4

Experimental Analysis

This section will summarize the key findings and results of the analysis. The aim of our analysis is to compare and find the best performing models that can be used in general despite class imbalance and dataset size. First we will explore the features of each dataset and the preparations needed to run the models, then we will look at how to tune the parameters for each model to get better performance and finally we will discuss the results of our experiment and rank the models.

4.1 Computing environment

Before going into the analysis, it is important to mention the computing environment in which the analysis has been done as well as the programming language and libraries used.

The code is written using python programming language which can be found at `https://www.python.org/`.

Library	Version
Python	Python 3.11.5
sklearn	1.5.1
scipy	1.13.1
pandas	2.2.2
numpy	1.23.5
matplotlib	3.8.4

Table 4.1: Software versions

4.2 Ethical Considerations

As credit scoring deals with sensitive banking client information, it is important that all personally identifying information is removed from the datasets before performing any analysis.

Analysts must also make sure to comply with data protection regulations like GDPR.

4.3 Exploratory Data Analysis

4.3.1 Dataset description

Expanding on the abstract (0.1), Table 4.2 shows some summarizing information on the datasets.

Dataset	No of rows	No of feat	Has Missing	Imbalance
Australian Credit (AC)	690	14	No	1.25
Give Me Some Credit (GMSC)	150000	12	Yes	13.96
German Credit (GC)	1000	20	No	2.33
Taiwan Credit Card (TC)	30000	25	No	3.52
Dataset	No of Cat	No of Bin	No of Numeric	
Australian Credit (AC)	3	4	7	
Give Me Some Credit (GMSC)	0	0	10	
German Credit (GC)	12	4	3	
Taiwan Credit Card (TC)	8	1	15	

Table 4.2: Datasets Overview

The imbalance here describes the number of non defaults (class 0) as a ratio to defaults (class 1). Table 4.2 shows that GMSC has the highest class imbalance at a ratio of 13.96, meaning that for every 14 non defaults, there is only 1 default. The second highest is TC but the imbalance is not as extreme in that dataset. AC however, had a balanced distribution. As mentioned in 3.5.1 the imbalance will cause metrics like accuracy to be artificially high. Hence why other metrics which are less affected by this imbalance are used in our evaluations.

We also notice that the size of each dataset is very different. We have a good mix of large, small, and medium sized ones. This will affect the suitability of models as some models like SVM, as mentioned in 3.2.3, scale poorly with the number of observations.

4.3.2 GMSC

Table 4.3 reveals the features with missing values and how many of them there are.

	count	mean	std	min	25%	50%	75%	max
RevolvingUtilization OfUnsecuredLines	150000	6.05	249.76	0.00	0.03	0.15	0.56	50708.00
age	150000	52.30	14.77	0.00	41.00	52.00	63.00	109.00
NumberOfTime30-59 DaysPastDueNotWorse	150000	0.42	4.19	0.00	0.00	0.00	0.00	98.00
DebtRatio	150000	353.01	2037.82	0.00	0.18	0.37	0.87	329664.00
MonthlyIncome	120269	6670.22	14384.67	0.00	3400.00	5400.00	8249.00	3008750.00
NumberOfOpenCredit LinesAndLoans	150000	8.45	5.15	0.00	5.00	8.00	11.00	58.00
NumberOfTimes90DaysLate	150000	0.27	4.17	0.00	0.00	0.00	0.00	98.00
NumberRealEstateLoansOrLines	150000	1.02	1.13	0.00	0.00	1.00	2.00	54.00
NumberOfTime60-89 DaysPastDueNotWorse	150000	0.24	4.16	0.00	0.00	0.00	0.00	98.00
NumberOfDependents	146076	0.76	1.12	0.00	0.00	0.00	1.00	20.00

Table 4.3: GMSC description

There are 29731 missing MonthlyIncome and 3924 missing NumberOfDependents. Since these features both have some extreme values, a mean imputation would be skewed towards the higher side. Hence, we impute these values with the median.

We notice that the age variable has a minimum of 0. This is not possible, and we need to make sure that the minimum age is 18 which reflects the law. There was only one observation with this issue and it was also replaced with the median.

To get an idea of the interaction between the dependent variables we also calculate the correlation between all of them.

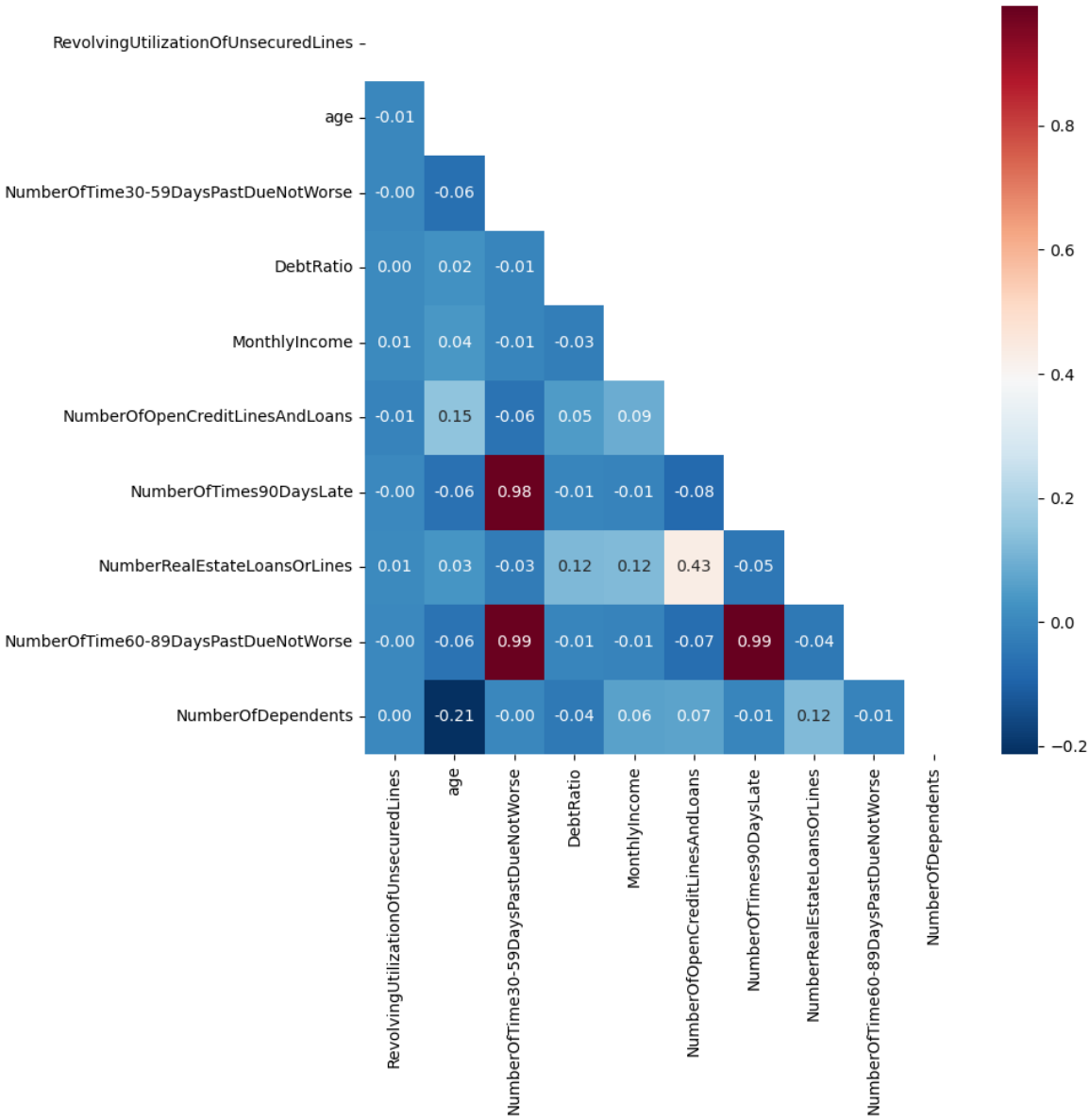


Figure 4.1: GMSC cor plot

Figure 4.1 shows the correlations between the variables. Especially concerning is 3 variables that have very high correlation. Preliminary intuition would dictate that 2 of these can be safely removed. These features in particular are NumberOfTime30-59DaysPastDueNotWorse, NumberOfTime60-89DaysPastDueNotWorse, NumberOfTimes90DaysLate. These tell the number of times they have been x days late to payments in the past in 2 years. Further inspection reveals that these 3 variables have a small cluster of values of 96 and 98. These values are much higher than the remaining and it is not possible to have been late for 30 days more than 24 times in the last 2 years. Hence, these values are either code for a missing value or a mistake in the data input. The values of 96 and 98 are thus replaced with the median.

```
[124]: for col in ['NumberOfTime30-59DaysPastDueNotWorse', 'NumberOfTime60-89DaysPastDueNotWorse', 'NumberOfTimes90DaysLate']:
           display(examinePercentiles(Xs[1][col], 0.99,  0.001))
```

	Percentile	# Examples to Percentile	# Examples in bin	Example Level
0	0.991	149150.0	149150.0	4.0
1	0.992	149150.0	0.0	4.0
2	0.993	149150.0	0.0	4.0
3	0.994	149150.0	0.0	4.0
4	0.995	149492.0	342.0	5.0
5	0.996	149492.0	0.0	5.0
6	0.997	149632.0	140.0	6.0
7	0.998	149711.0	79.0	8.0
8	0.999	150000.0	289.0	98.0
9	1.000	150000.0	0.0	98.0

Figure 4.2: Quantiles from the 99th percentile

Figure 4.2 shows the quantiles of NumberOfTime30-59DaysPastDueNotWorse from the 99th percentile and up. We can see a sudden disconnect between 0.998 and 0.999.

Following the imputation, we calculate the correlations again to get a very different picture.

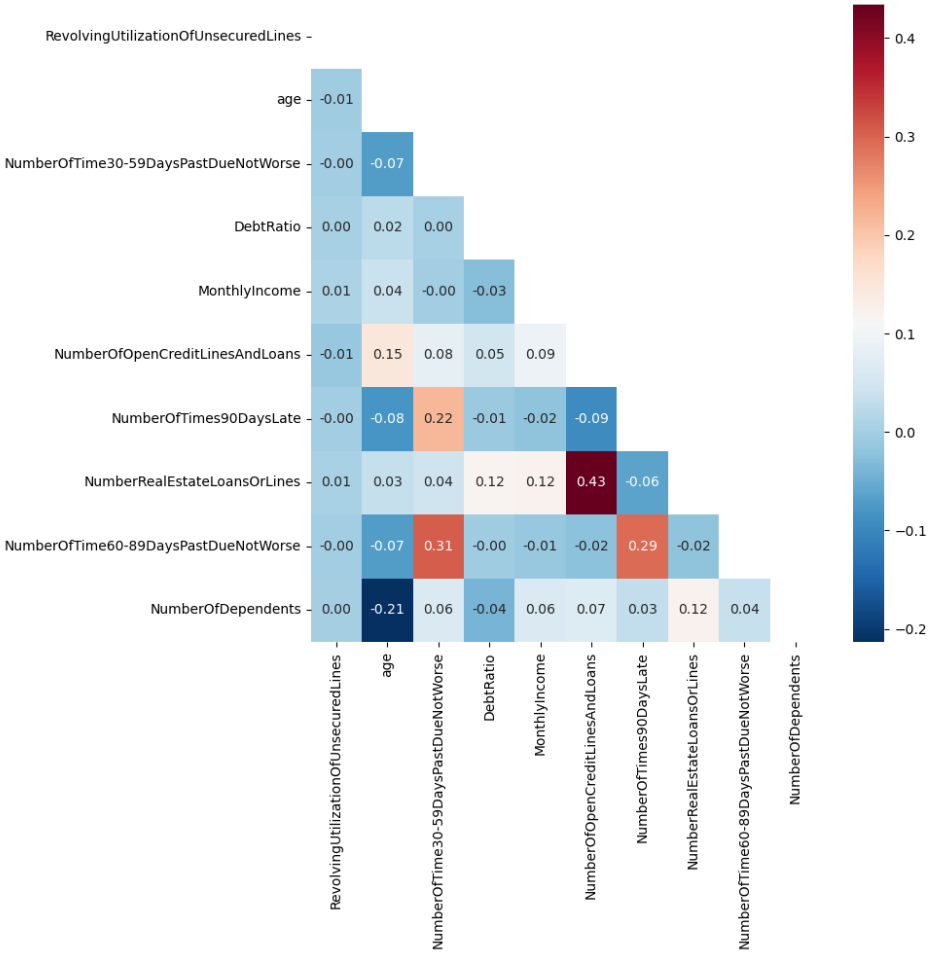


Figure 4.3: GMSC cor plot after imputation

Figure 4.3 shows that these features are now no longer very correlated.

4.3.3 Taiwan credit

TC had a single major issue. Other than multiple outliers, there were multiple highly correlated variables, as illustrated by Figure 4.4.

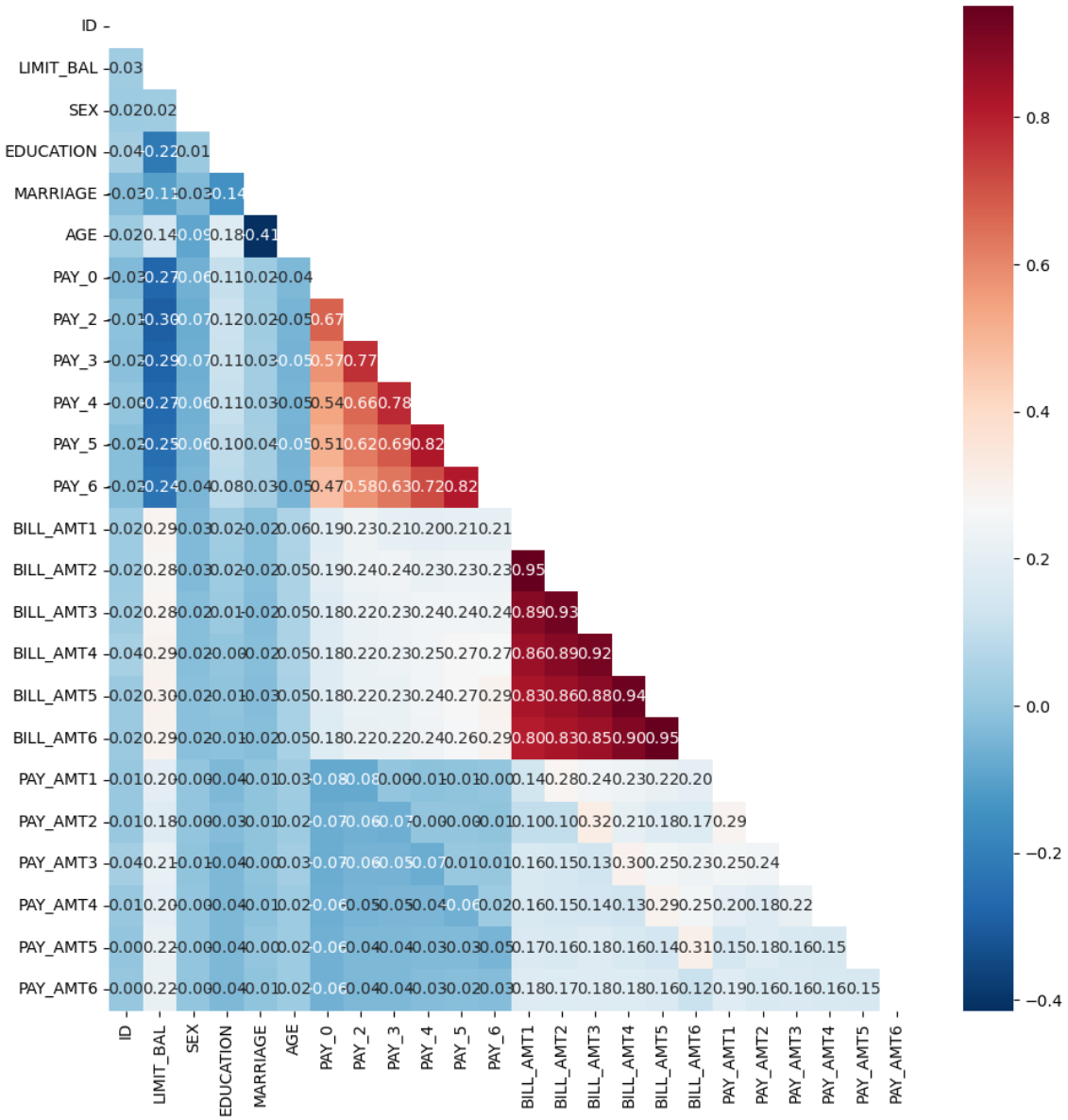


Figure 4.4: TC cor plot

These could be removed so as to prevent over-fitting and reduce training time. These variables include PAY_2 to PAY_6 which shows that after the second month, credit repayment habits remain the same. The BILL_AMOUNT variables are also highly correlated indicating that bill amounts increase over time.

We can also look at the distribution of the numeric variables in TC using histograms to get an idea of suitable scaling methods.

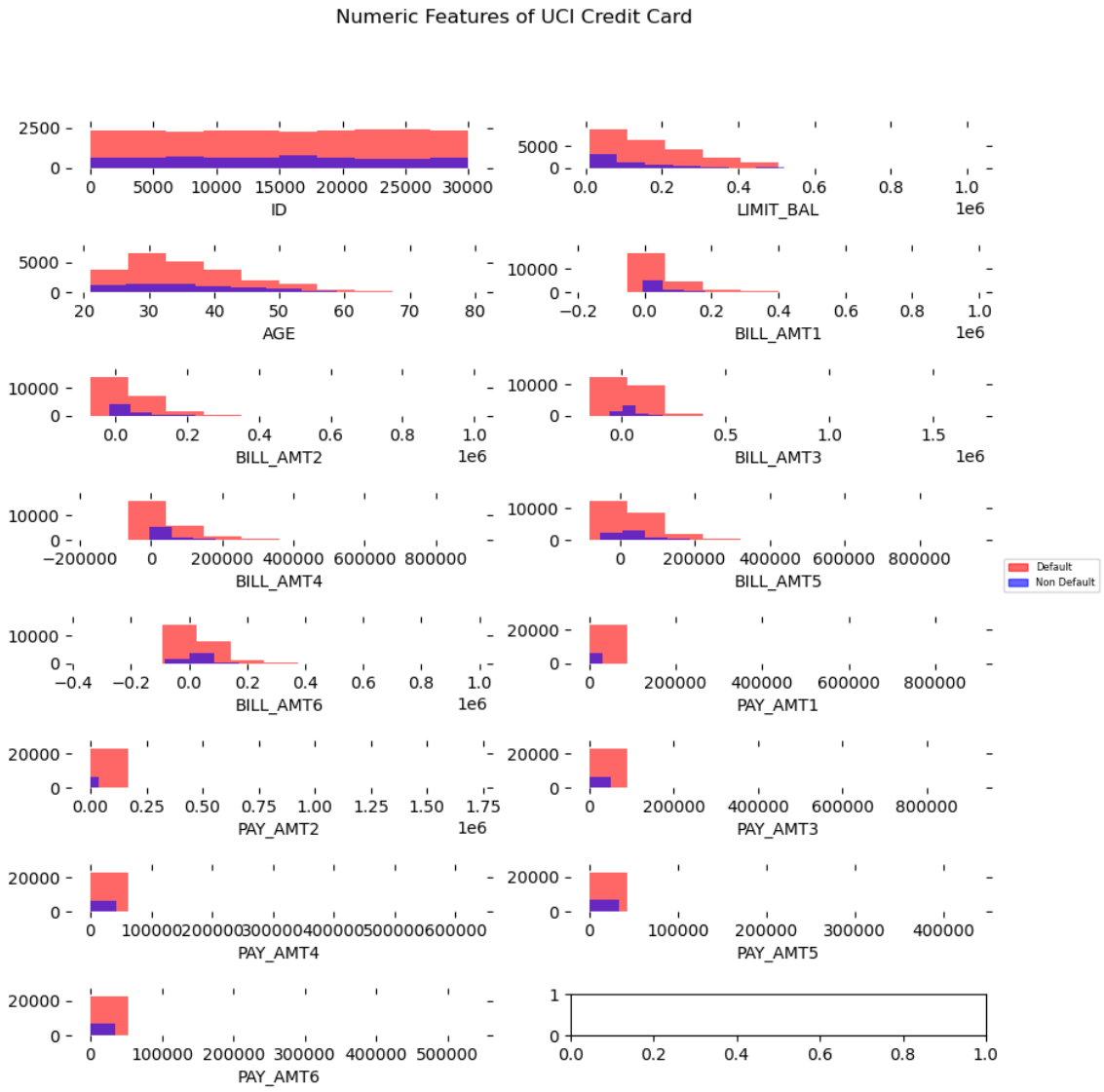


Figure 4.5: TC histograms

From Figure 4.5, looking at PAY_AMT1 we notice that the x axis range is large, while most observations are concentrated on the left. This indicates the presence of outliers that would influence the standardization. The age variable is normally distributed and shows a correct cut off below 21 years old. The ID variable is redundant here and can safely be removed.

4.3.4 Australian Credit and German Credit

These datasets didn't require any particular feature selection or imputations. For AC, we do not have information on the variable names but only whether they are categorical or numeric. After plotting the distributions of the categorical variables, we can see that there are some features where some categories actually have more defaults than non defaults. We can see these as the dataset is balanced.

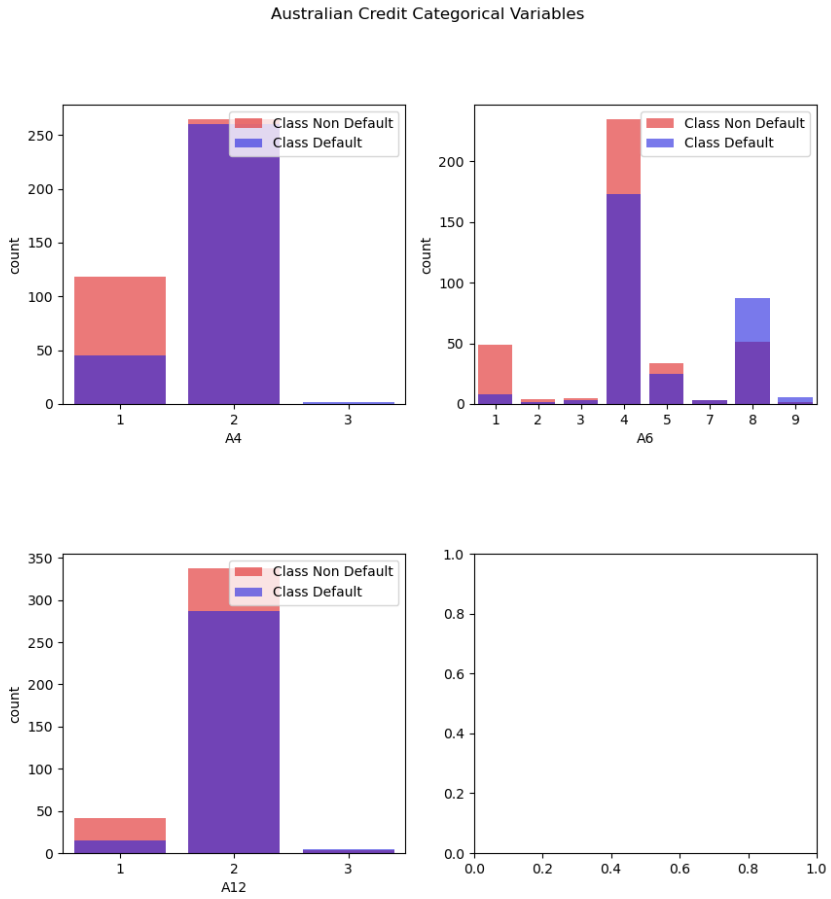


Figure 4.6: AC categorical plot

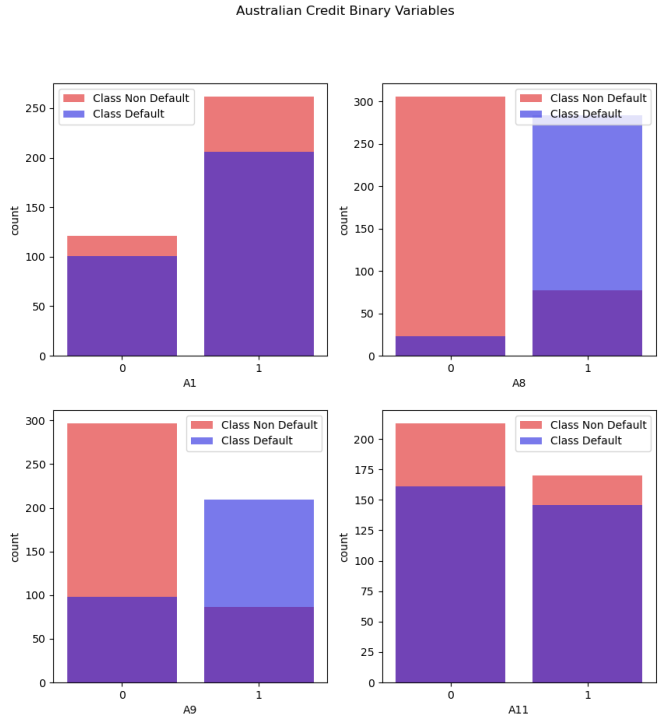


Figure 4.7: AC binary plot

Figures 4.7 and 4.6 respectively show the distribution of categorical and binary variables in AC. For instance A8 has an equal number of category 1 and 2 and these have opposite defaulting trends. A6 starts to get higher defaults than non defaults as from category 8.

Additional plots and tables on the datasets can be found in the appendix (B.1)

4.4 Pipelines

So as to prevent leaking of information from the test set (validating set) and the training set, we cannot apply all transformations to the whole dataset. This is because for transformations like standardization, the range of values available dictate the mean and variance to be used. Since in a real word scenario, we cannot get the mean and variance of all possible samples, we need to take this into consideration and only calculate the mean and variance of the training set and apply these same parameters when standardizing the testing set. The same is true for models. They are trained on the training set and only applied on the testing set. An efficient way to do this is using pipelines in sklearn. Pipeline allows you to sequentially apply a list of transformers to preprocess the data and, if desired, conclude the sequence with a final predictor for predictive modeling. All that is needed is to specify the operations to be applied and their order.

```
ct1 = ColumnTransformer([
    ("impute_and_scale", impute_and_scale, ['MonthlyIncome', 'NumberOfDependents']),
    ("standardised", StandardScaler(), ["age"]),
    ("robust", RobustScaler(), ['RevolvingUtilizationOfUnsecuredLines',
       'NumberOfTime30-59DaysPastDueNotWorse', 'DebtRatio',
       'NumberOfOpenCreditLinesAndLoans',
       'NumberRealEstateLoansOrLines', 'NumberOfTime60-89DaysPastDueNotWorse', 'NumberOfTimes90DaysLate'])
])
```

Figure 4.8: column transformations for GMSC

And we combine 4.8 with the pipeline to train an MLP:

```
Pipeline1_mlp = Pipeline(steps=[("preprocessor", ct1), ("classifier",
↪  MLPClassifier(random_state=7, early_stopping=True))])
```

4.5 Hyperparameter tuning

Bergstra and Bengio, 2012 defines hyperparameter tuning as the parameters that are set before a function f that minimizes an expected loss to learn from a training data decides on the optimal parameters on the model. Hence, by optimizing the hyperparameters, we can influence the way a model works and the parameters that it generates from the training data. An example of a hyperparameter is the regularization penalty C in SVM and the number of trees in RF.

Model	AC
Logistic Regression	C=1, solver=liblinear
MLP	hidden_layer_sizes=(200,100), max_iter=2000, early_stopping=True, random_state=7
Random Forest	max_depth=30, max_features=log2, min_samples_split=5
KNN	leaf_size=40, n_neighbors=15, weights=distance
Decision Tree	max_depth=10, max_features=sqrt, min_samples_leaf=6, random_state=7
SVM	C=10, max_iter=10000, random_state=7, probability=True
LDA	default

Table 4.4: Hyperparameters used for AC

Model	GMSC
Logistic Regression	C=5, solver=newton-cg, max_iter=500, random_state=7
MLP	hidden_layer_sizes=(200,), max_iter=2500, activation=logistic, early_stopping=True, random_state=7
Random Forest	max_features=log2, min_samples_split=10
KNN	leaf_size=40, n_neighbors=3
Decision Tree	max_features=sqrt, min_samples_leaf=6, random_state=7
SVM	C=0.1, max_iter=10000, random_state=7
LDA	default

Table 4.5: Hyperparameters used for GMSC

Model	GC
Logistic Regression	C=1, solver=newton-cg
MLP	hidden_layer_sizes=(200,100), max_iter=2000, activation=tanh, solver=lbfgs, random_state=7
Random Forest	max_depth=30, min_samples_leaf=2, min_samples_split=10
KNN	leaf_size=20, n_neighbors=3, weights=distance
Decision Tree	max_depth=10, max_features=sqrt, min_samples_leaf=6, random_state=7
SVM	C=10, max_iter=10000, random_state=7, probability=True
LDA	default

Table 4.6: Hyperparameters used for GC

Model	TC
Logistic Regression	C=10, solver=newton-cg, max_iter=500, random_state=7
MLP	hidden_layer_sizes=(200,100), max_iter=2500, early_stopping=True, random_state=7
Random Forest	max_depth=30, min_samples_split=10, random_state=7
KNN	algorithm=ball_tree, leaf_size=40, n_neighbors=10, weights=distance
Decision Tree	max_depth=10, max_features=sqrt, random_state=7
SVM	C=0.1, max_iter=10000, random_state=7
LDA	default

Table 4.7: Hyperparameters used for TC

4.6 Results

Finally, we can analyse the results from running the models.

After averaging the scores over all 10 cross validations for each model, we get the following test results (4.8).

		fit_time	score_time	acc	precision	recall	ks	kappa	roc_auc
AC	lr	0.03	0.04	0.79	0.82	0.69	0.63	0.57	0.86
	rf	0.24	0.04	0.81	0.83	0.74	0.66	0.61	0.88
	lda	0.02	0.03	0.77	0.80	0.63	0.59	0.52	0.84
	knn	0.02	0.05	0.77	0.83	0.61	0.58	0.52	0.84
	mlp	0.24	0.03	0.80	0.85	0.66	0.64	0.58	0.86
	dt	0.02	0.03	0.73	0.74	0.63	0.48	0.45	0.77
	svm	0.23	0.05	0.80	0.88	0.65	0.64	0.59	0.87
GMSC	lr	2.28	0.07	0.93	0.55	0.01	0.24	0.03	0.66
	rf	54.27	0.78	0.93	0.52	0.10	0.51	0.14	0.82
	lda	0.45	0.07	0.93	0.55	0.01	0.27	0.03	0.69
	knn	0.61	8.10	0.92	0.32	0.13	0.27	0.15	0.64
	mlp	28.63	0.21	0.93	0.54	0.05	0.49	0.08	0.80
	dt	0.70	0.06	0.93	0.36	0.14	0.36	0.17	0.69
	svm	1.43	0.07	0.93	0.55	0.01	0.28	0.03	0.69
GC	lr	0.05	0.05	0.75	0.62	0.47	0.49	0.37	0.77
	rf	0.26	0.05	0.76	0.70	0.34	0.51	0.32	0.79
	lda	0.04	0.04	0.75	0.61	0.50	0.48	0.37	0.77
	knn	0.02	0.05	0.72	0.54	0.39	0.37	0.26	0.69
	mlp	1.51	0.03	0.71	0.52	0.50	0.40	0.30	0.72
	dt	0.02	0.03	0.69	0.48	0.33	0.33	0.19	0.68
	svm	0.70	0.07	0.74	0.58	0.53	0.42	0.37	0.75
TC	lr	0.80	0.05	0.82	0.70	0.32	0.39	0.35	0.75
	rf	6.98	0.18	0.82	0.68	0.35	0.41	0.37	0.77
	lda	0.38	0.05	0.82	0.70	0.33	0.39	0.35	0.75
	knn	0.26	26.21	0.80	0.61	0.32	0.34	0.31	0.72
	mlp	10.52	0.07	0.82	0.69	0.33	0.41	0.35	0.77
	dt	0.11	0.04	0.81	0.63	0.30	0.35	0.31	0.73
	svm	0.52	0.04	0.82	0.70	0.33	0.39	0.35	0.75

Table 4.8: All Scores

There are too many metrics to interpret them individually but we can pick a few that are worth exploring.

The recall percentages are very low for GMSC but surprisingly, KNN performed very well in

comparison. As expected the tree based algorithms also did well in the face of class imbalance. In comparison the accuracy scores are all very high. That is because the model simply optimised to predict non defaults and it would be correct most of the time as 93% of the classes were non defaults. As for the others AC had 56% of non defaults, GC had 70% and TC had 78%. Taking this into account, we can see that the models mostly had an accuracy score greater or equal to the percentage of majority class in the dataset.

Looking at the fit time scores from the cross_validate class in sklearn, we notice that the models that took longest to train were MLP, SVM with non-linear kernel, and RF. All the models also performed best on the AC dataset, despite having relatively few observations, the balanced dataset meant that the models were more capable of classifying the target correctly.

4.7 Ranking

To get a more holistic overview of the model performances, an intuitive way to achieve this goal is to rank the models and compare their ranks.

Two approaches have been used to get the average rank the models but they led to the same interpretation.

1. A simple mean of ranks
2. A weighted mean giving more importance to recall and less to the other metrics.

	AVG Rank						
	lr	rf	lda	knn	mlp	dt	svm
AC	3.83	1.5	5.67	5.17	2.67	6.83	2.33
GMSC	5.5	2.33	4.00	5.17	3.00	3.33	3.67
GC	2.5	2.5	2.5	6	5	7	2.5
TC	2.83	2.33	3.5	6.83	3.33	6.17	3
Total	14.66	8.66	15.67	23.17	14.00	23.33	11.50

Table 4.9: Average Ranks

From Table 4.9 we can deduce that RF performed the best overall performance, followed by SVM, MLP, LR, LDA, KNN and finally DT. The poor performance of DT is interesting to notice, and it was always among the last ranking on all datasets.

For the weighted average, the weights have been distributed as follows:

$$0.3 \times recall + 0.2 \times auc + 0.15 \times ks + 0.15 \times precision + 0.1 \times \kappa + 0.1 \times accuracy$$

	Weighted Rank						
	lr	rf	lda	knn	mlp	dt	svm
AC	3.55	1.45	5.55	5.45	2.7	6.7	2.6
GMSC	5.4	2.4	4.2	4.85	3.1	2.95	3.85
GC	2.7	2.8	2.9	5.45	4.35	7	2.8
TC	3.2	2.05	3.6	6.7	3.25	6.3	2.9
Total	14.85	8.70	16.25	22.45	13.40	22.95	12.15

Table 4.10: Weighted Ranks

4.8 Testing signficance of results

Under the null hypothesis that the repeated samples of the weighted model ranks have the same distribution across the four datasets against the alternative hypothesis that they are different, we can do a friedman test at the 5% level. We would reject H_0 if the p value of the χ^2 statistic is less than 0.05.

```
[4]: from scipy.stats import friedmanchisquare
     ac_ranks = [3.55,1.45,5.55,5.45,2.7,6.7,2.6]
     gmsc_ranks = [5.4,2.4,4.2,4.85,3.1,2.95,3.85]
     gc_ranks = [2.7,2.8,2.9,5.45,4.35,7,2.8]
     tc_ranks = [3.2,2.05,3.6,6.7,3.25,6.3,2.9]

     res = friedmanchisquare(ac_ranks, gmsc_ranks, gc_ranks, tc_ranks)
     if res.pvalue < 0.05:
         print(f"The X2 statistic is {res.statistic:.2f} with a p value = {res.pvalue:.2f}, we reject H0")
     else:
         print(f"The X2 statistic is {res.statistic:.2f} with a p value = {res.pvalue:.2f}, we fail to reject H0")

     The X2 statistic is 0.48 with a p value = 0.92, we fail to reject H0
```

Figure 4.9: Friedman test

We have a very large p value so we can say that the rankings are statistically significant.

Chapter 5

Conclusion

In this study we did a comprehensive analysis of various machine learning models for credit scoring across four different datasets. The aim of this study was to determine which model performed the best consistently in different situations such as class imbalance, different dataset sizes and different types of features. In so doing, we evaluated seven machine learning models and compared their performance metrics.

We began by doing Exploratory Data Analysis (EDA) on the datasets to identify any potential issues that could arise when applying the models. Together with the data cleaning and data transformation step this could be considered the most important step in the machine learning pipeline. These preliminary analyses gave us information on what the best normalization and imputation procedures would be as well as what features could be removed to decrease the complexity of the models.

The hyper-parameter tuning process was also very important and while time consuming especially for larger models and testing for multiple parameters across multiple cross validation folds, it can improve model performance and ensures that they are well adjusted to the specific dataset.

To mitigate the issues of class imbalance, cross validation with stratified sampling was used to maintain the representation of the class ratios in the training and testing sets. However, despite this we noted that the larger the imbalance in the dataset, the less the models were capable of correctly classifying the minority classes. This means that in cases of class imbalance, a sampling technique like up sampling of the minority class and down sampling of the majority class is of utmost importance to get good performance results.

The RF was the best performing model overall, being capable of handling different scenarios well while also being relatively easy to implement. A close second was SVM which also came close to the performance of RF. Conversely, we found that DT was consistently among the worst performing models.

In the future, a more thorough exploratory data analysis can be done to explain the features

that could be most useful in the models. Individual model performance may be improved by using more complicated imputation methods, implementing more robust feature selection and feature engineering, using sampling techniques to combat class imbalance and applying modern ensemble methods.

References

1.11. Ensembles (2024). *1.11. Ensembles: Gradient Boosting, Random Forests, Bagging, Voting, Stacking*. scikit-learn. URL: https://scikit-learn/stable/modules/ensemble.html (visited on 09/20/2024).

1.17. Neural Network Models (Supervised) (2024). scikit-learn. URL: https://scikit-learn/stable/modules/neural_networks_supervised.html (visited on 09/16/2024).

1.4. Support Vector Machines (2024). scikit-learn. URL: https://scikit-learn/stable/modules/svm.html (visited on 09/22/2024).

3.1. Cross-validation (2024). *3.1. Cross-validation: Evaluating Estimator Performance*. scikit-learn. URL: https://scikit-learn/stable/modules/cross_validation.html (visited on 09/21/2024).

Abdou, Hussein A. and John Pointon (2011). "Credit Scoring, Statistical Techniques and Evaluation Criteria: A Review of the Literature". In: *Intelligent Systems in Accounting, Finance and Management* 18.2-3, pp. 59–88. ISSN: 1099-1174. DOI: 10.1002/isaf.325. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/isaf.325 (visited on 03/01/2024).

Addy, Wilhelmina et al. (Feb. 20, 2024). "AI in Credit Scoring: A Comprehensive Review of Models and Predictive Analytics". In: 18, pp. 118–129. DOI: 10.30574/gjeta.2024.18.2.0029.

Al Jallad, Khloud, Said Desouki, and Mohamad Aljnidi (Aug. 2020). "Anomaly Detection Optimization Using Big Data and Deep Learning to Reduce False-Positive". In: *Journal of Big Data* 7. DOI: 10.1186/s40537-020-00346-1.

Bacallado, Sergio and Jonathan Taylor (2024). *Linear Discriminant Analysis (LDA) — STATS 202*. URL: https://web.stanford.edu/class/stats202//notes/Classification/LDA.html (visited on 09/05/2024).

Baesens, B et al. (2003). "Benchmarking State-of-the-Art Classification Algorithms for Credit Scoring". In: *Journal of the Operational Research Society* 54.6, pp. 627–635. DOI: 10.1057/palgrave.jors.2601545. eprint: https://doi.org/10.1057/palgrave.jors.2601545. URL: https://doi.org/10.1057/palgrave.jors.2601545.

Bergstra, James and Yoshua Bengio (Feb. 1, 2012). "Random Search for Hyper-Parameter Optimization". In: *J. Mach. Learn. Res.* 13 (null), pp. 281–305. ISSN: 1532-4435.

Breiman, Leo (Oct. 1, 2001). "Random Forests". In: *Machine Learning* 45.1, pp. 5–32. ISSN: 1573-0565. DOI: 10.1023/A:1010933404324. URL: https://doi.org/10.1023/A:1010933404324 (visited on 09/20/2024).

Brown, Iain and Christophe Mues (Feb. 15, 2012). "An Experimental Comparison of Classification Algorithms for Imbalanced Credit Scoring Data Sets". In: *Expert Systems with Applications* 39.3, pp. 3446–3453. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2011.09.033. URL: https://www.sciencedirect.com/science/article/pii/S095741741101342X (visited on 09/02/2024).

Cortes, Corinna and Vladimir Vapnik (Sept. 1, 1995). "Support-Vector Networks". In: *Machine Learning* 20.3, pp. 273–297. ISSN: 1573-0565. DOI: 10.1007/BF00994018. URL: https://doi.org/10.1007/BF00994018.

Credit Fusion, Will Cukierski (2011). *Give Me Some Credit*. Kaggle. URL: https://kaggle.com/competitions/GiveMeSomeCredit (visited on 02/22/2024).

Crook, Jonathan (1996). "Credit Scoring: An Overview". In: *European Journal of Operational Research* 95, pp. 24–37. URL: https://www.research.ed.ac.uk/en/publications/credit-scoring-an-overview-2 (visited on 08/26/2024).

Crook, Jonathan N., David B. Edelman, and Lyn C. Thomas (Dec. 16, 2007). "Recent Developments in Consumer Credit Risk Assessment". In: *European Journal of Operational Research* 183.3, pp. 1447–1465. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2006.09.100. URL: https://www.sciencedirect.com/science/article/abs/pii/S0377221706011866 (visited on 08/01/2024).

Dubey, Shiv Ram, Satish Kumar Singh, and Bidyut Baran Chaudhuri (June 28, 2022). *Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark*. arXiv: 2109.14545 [cs]. URL: http://arxiv.org/abs/2109.14545 (visited on 09/16/2024). Pre-published.

Fawcett, Tom (June 1, 2006). "An Introduction to ROC Analysis". In: *Pattern Recognition Letters*. ROC Analysis in Pattern Recognition 27.8, pp. 861–874. ISSN: 0167-8655. DOI: 10.1016/j.patrec.2005.10.010. URL: https://www.sciencedirect.com/science/article/pii/S016786550500303X (visited on 06/05/2024).

Hand, D. J. and W. E. Henley (Sept. 1, 1997). "Statistical Classification Methods in Consumer Credit Scoring: A Review". In: *Journal of the Royal Statistical Society Series A: Statistics in Society* 160.3, pp. 523–541. ISSN: 0964-1998, 1467-985X. DOI: 10.1111/j.1467-985X.1997.00078.x. URL: https://academic.oup.com/jrsssa/article/160/3/523/7102381 (visited on 08/05/2024).

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman (2009). *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY: Springer. ISBN: 978-0-387-84857-0. DOI: 10.1007/978-0-387-84858-7. URL: http://link.springer.com/10.1007/978-0-387-84858-7 (visited on 07/22/2024).

Hofmann, Hans (1994). *Statlog (German Credit Data)*. UCI Machine Learning Repository. DOI: 10.24432/C5NC77. URL: https://archive.ics.uci.edu/dataset/144 (visited on 01/30/2024).

I-Cheng Yeh (2009). *Default of Credit Card Clients*. UCI Machine Learning Repository. DOI: 10.24432/C55S3H. URL: https://archive.ics.uci.edu/dataset/350 (visited on 09/02/2024).

Jurafsky, Daniel and James H Martin (2024). *5 - Notes - Speech and Language Processing. Daniel Jurafsky & James H. Martin. Copyright © 2023. - Studocu*. URL: https://www.studocu.com/in/document/manipal-university-jaipur/dsml/5-notes/52554292 (visited on 09/02/2024).

Lessmann, Stefan et al. (Nov. 16, 2015). "Benchmarking State-of-the-Art Classification Algorithms for Credit Scoring: An Update of Research". In: *European Journal of Operational Research* 247.1, pp. 124–136. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2015.05.030. URL: https://www.sciencedirect.com/science/article/pii/S0377221715004208 (visited on 03/01/2024).

Quinlan, Ross (1987). *Statlog (Australian Credit Approval)*. [object Object]. DOI: 10.24432/C59012. URL: https://archive.ics.uci.edu/dataset/143 (visited on 05/11/2024).

Rice, John A. (1988). "Mathematical Statistics and Data Analysis". In: URL: https://api.semanticscholar.org/CorpusID:260874435.

RobustScaler (2024). scikit-learn. URL: https://scikit-learn/stable/modules/generated/sklearn.preprocessing.RobustScaler.html (visited on 09/21/2024).

Stelzer, Anna (July 30, 2019). *Predicting Credit Default Probabilities Using Machine Learning Techniques in the Face of Unequal Class Distributions*. arXiv: 1907.12996 [cs, econ]. URL: http://arxiv.org/abs/1907.12996 (visited on 03/01/2024). Pre-published.

Wainer, Jacques and Gavin Cawley (2021). "Nested Cross-Validation When Selecting Classifiers Is Overzealous for Most Practical Applications". In: *Expert Systems with Applications* 182, p. 115222. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2021.115222. URL: https://www.sciencedirect.com/science/article/pii/S0957417421006540.

Weng, Cheng-Hsiung and Cheng-Kui Huang (Dec. 15, 2021). "A Hybrid Machine Learning Model for Credit Approval". In: *Applied Artificial Intelligence* 35.15, pp. 1439–1465. ISSN: 0883-9514. DOI: 10.1080/08839514.2021.1982475. URL: https://doi.org/10.1080/08839514.2021.1982475 (visited on 03/04/2024).

What Is Support Vector Machine? (Dec. 12, 2023). *What Is Support Vector Machine? | IBM*. URL: https://www.ibm.com/topics/support-vector-machine (visited on 09/22/2024).

Yeh, I-Cheng and Che-hui Lien (2009). "The Comparisons of Data Mining Techniques for the Predictive Accuracy of Probability of Default of Credit Card Clients". In: *Expert Systems With Applications* 36, pp. 2473–2480. URL: https://api.semanticscholar.org/CorpusID:15696161.

Appendix A

Tables

		fit_time	score_time	test_accuracy	train_accuracy	test_precision	train_precision	test_recall	train_recall	test_ks	train_ks	test_kappa	train_kappa	test_roc_auc	train_roc_auc
AC	lr	0.03	0.04	0.79	0.81	0.82	0.84	0.69	0.70	0.63	0.62	0.57	0.61	0.86	0.88
	rf	0.24	0.04	0.81	0.98	0.83	0.99	0.74	0.97	0.66	0.98	0.61	0.96	0.88	1.00
	lda	0.02	0.03	0.77	0.79	0.80	0.83	0.63	0.66	0.59	0.57	0.52	0.56	0.84	0.86
	knn	0.02	0.05	0.77	1.00	0.83	1.00	0.61	1.00	0.58	1.00	0.52	1.00	0.84	1.00
	mlp	0.24	0.03	0.80	0.81	0.85	0.86	0.66	0.68	0.64	0.62	0.58	0.60	0.86	0.88
	dt	0.02	0.03	0.73	0.82	0.74	0.85	0.63	0.74	0.48	0.64	0.45	0.64	0.77	0.91
	svm	0.23	0.05	0.80	0.82	0.88	0.90	0.65	0.67	0.64	0.64	0.59	0.63	0.87	0.88
GMSC	lr	2.28	0.07	0.93	0.93	0.55	0.54	0.01	0.01	0.24	0.24	0.03	0.03	0.66	0.66
	rf	54.27	0.78	0.93	0.96	0.52	0.99	0.10	0.40	0.51	0.97	0.14	0.56	0.82	1.00
	lda	0.45	0.07	0.93	0.93	0.55	0.55	0.01	0.01	0.27	0.26	0.03	0.03	0.69	0.69
	knn	0.61	8.10	0.92	0.95	0.32	0.72	0.13	0.32	0.27	0.90	0.15	0.42	0.64	0.96
	mlp	28.63	0.21	0.93	0.93	0.54	0.56	0.05	0.05	0.49	0.49	0.08	0.08	0.80	0.81
	dt	0.70	0.06	0.93	0.94	0.36	0.70	0.14	0.28	0.36	0.82	0.17	0.38	0.69	0.95
	svm	1.43	0.07	0.93	0.93	0.55	0.55	0.01	0.01	0.28	0.27	0.03	0.03	0.69	0.69
GC	lr	0.05	0.05	0.75	0.79	0.62	0.68	0.47	0.53	0.49	0.51	0.37	0.46	0.77	0.83
	rf	0.26	0.05	0.76	0.91	0.70	0.99	0.34	0.71	0.51	0.86	0.32	0.77	0.79	0.98
	lda	0.04	0.04	0.75	0.79	0.61	0.68	0.50	0.55	0.48	0.51	0.37	0.46	0.77	0.83
	knn	0.02	0.05	0.72	1.00	0.54	1.00	0.39	1.00	0.37	1.00	0.26	1.00	0.69	1.00
	mlp	1.51	0.03	0.71	1.00	0.52	1.00	0.50	1.00	0.40	1.00	0.30	1.00	0.72	1.00
	dt	0.02	0.03	0.69	0.79	0.48	0.72	0.33	0.50	0.33	0.53	0.19	0.46	0.68	0.85
	svm	0.70	0.07	0.74	0.99	0.58	0.99	0.53	0.98	0.42	0.99	0.37	0.98	0.75	1.00
TC	lr	0.80	0.05	0.82	0.82	0.70	0.70	0.32	0.32	0.39	0.39	0.35	0.35	0.75	0.75
	rf	6.98	0.18	0.82	0.90	0.68	0.94	0.35	0.59	0.41	0.92	0.37	0.67	0.77	0.99
	lda	0.38	0.05	0.82	0.82	0.70	0.70	0.33	0.33	0.39	0.38	0.35	0.35	0.75	0.75
	knn	0.26	26.21	0.80	1.00	0.61	1.00	0.32	1.00	0.34	1.00	0.31	1.00	0.72	0.75
	mlp	10.52	0.07	0.82	0.82	0.69	0.70	0.33	0.33	0.41	0.42	0.35	0.36	0.77	1.00
	dt	0.11	0.04	0.81	0.82	0.63	0.72	0.30	0.34	0.35	0.43	0.31	0.37	0.73	0.78
	svm	0.52	0.04	0.82	0.82	0.70	0.70	0.33	0.33	0.39	0.39	0.35	0.35	0.75	0.79

Table A.1: Train and test results

Appendix B

Figures

B.1 EDA

B.1.1 Binary variables

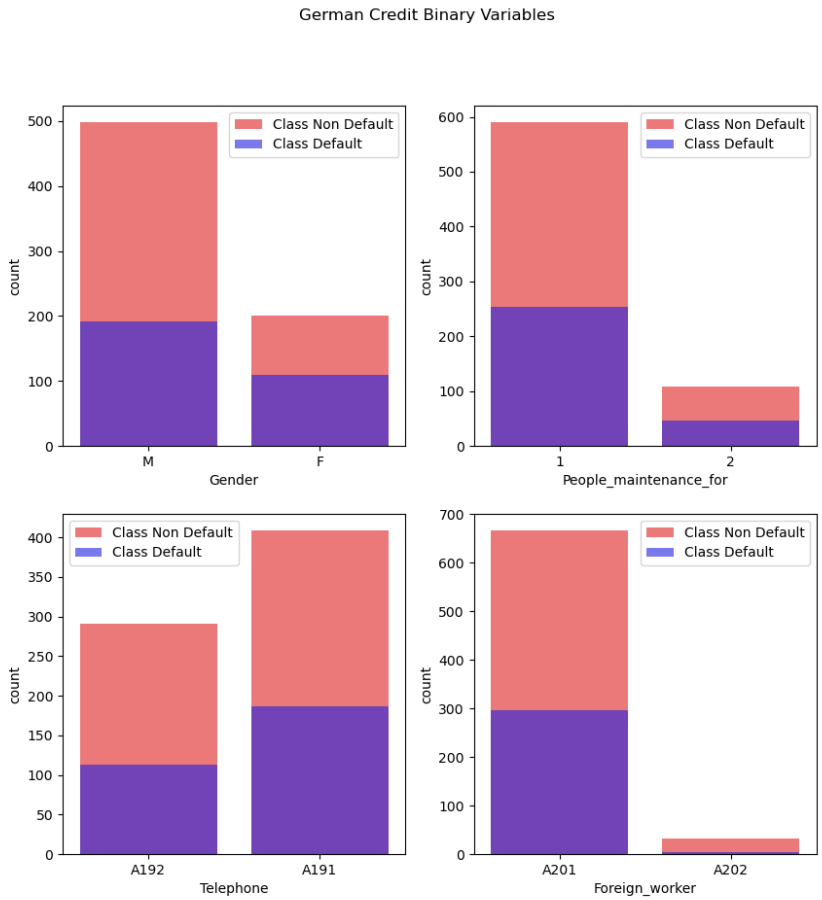


Figure B.1: GC binary plot

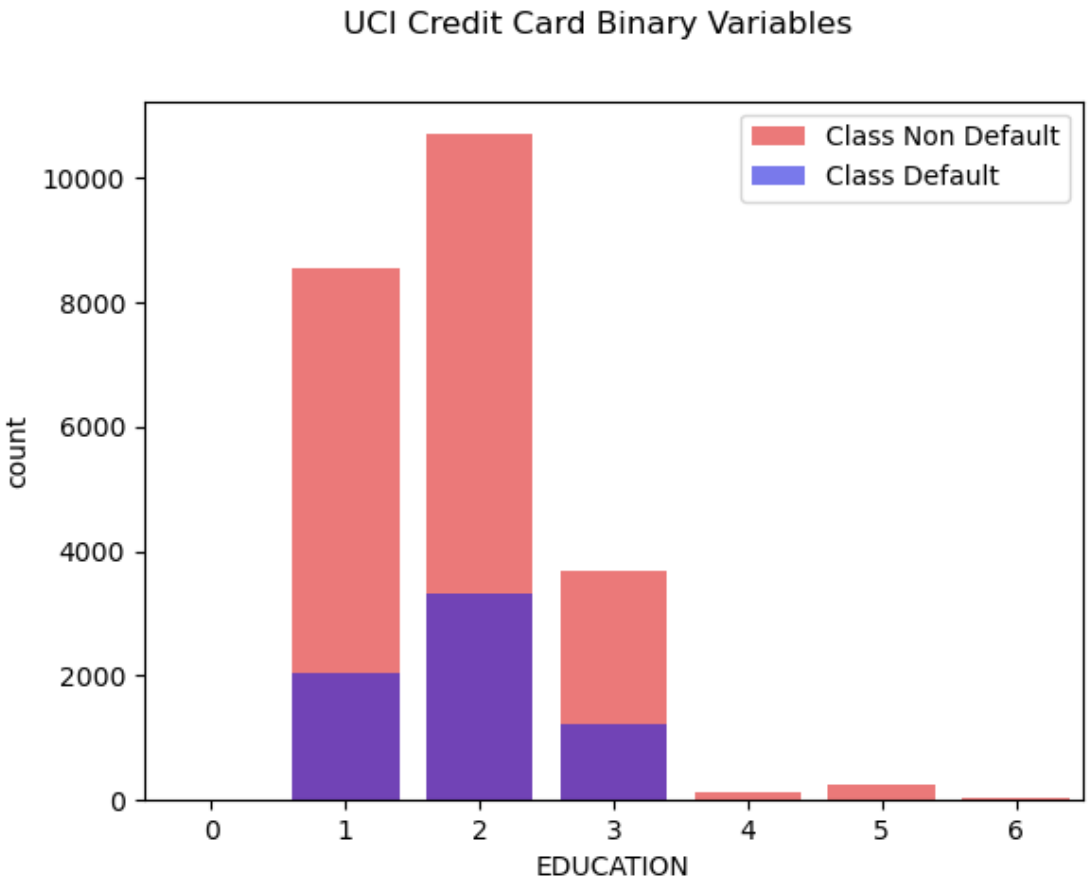


Figure B.2: TC binary plot

B.1.2 Categorical Variables

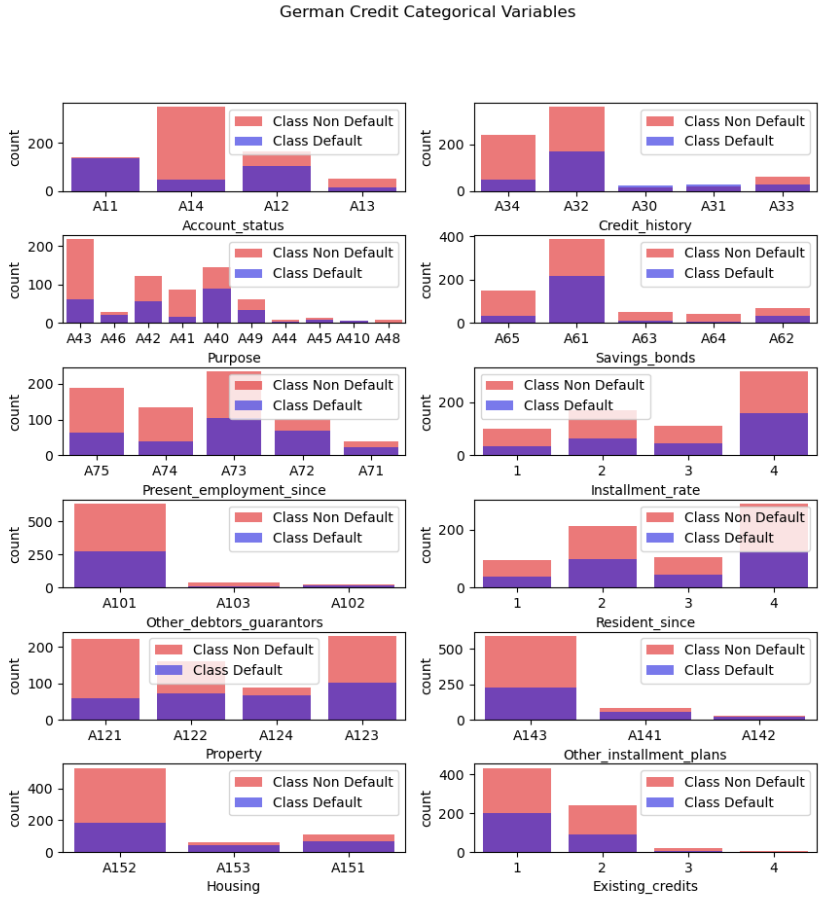


Figure B.3: GC categorical plot

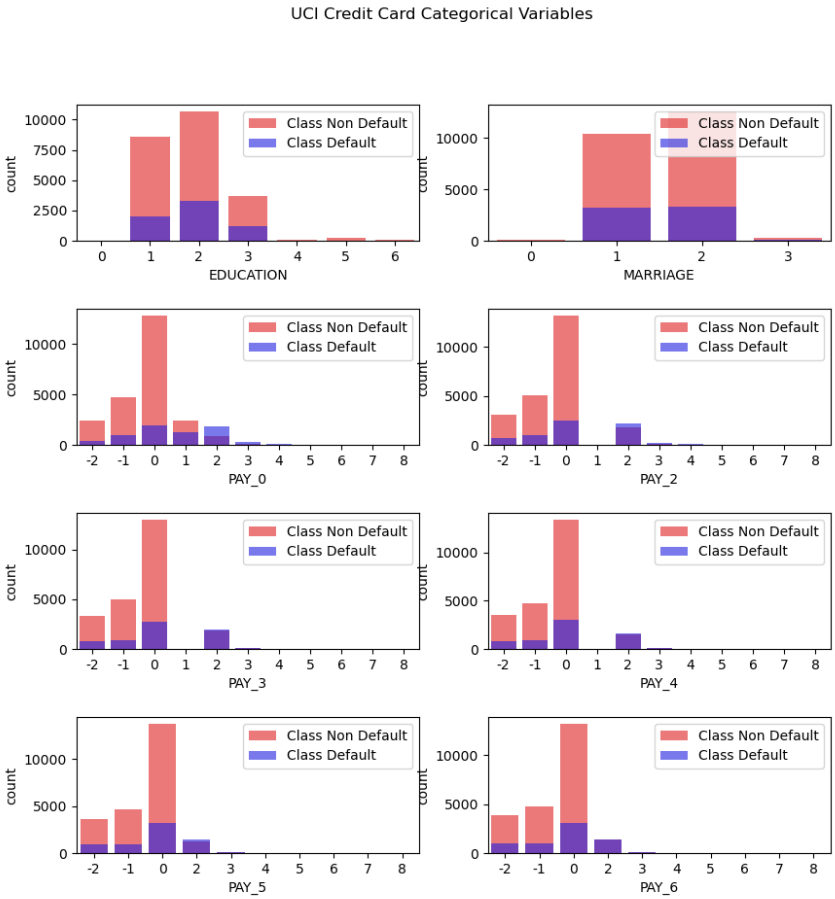


Figure B.4: TC categorical plot

B.2 Numeric Variables

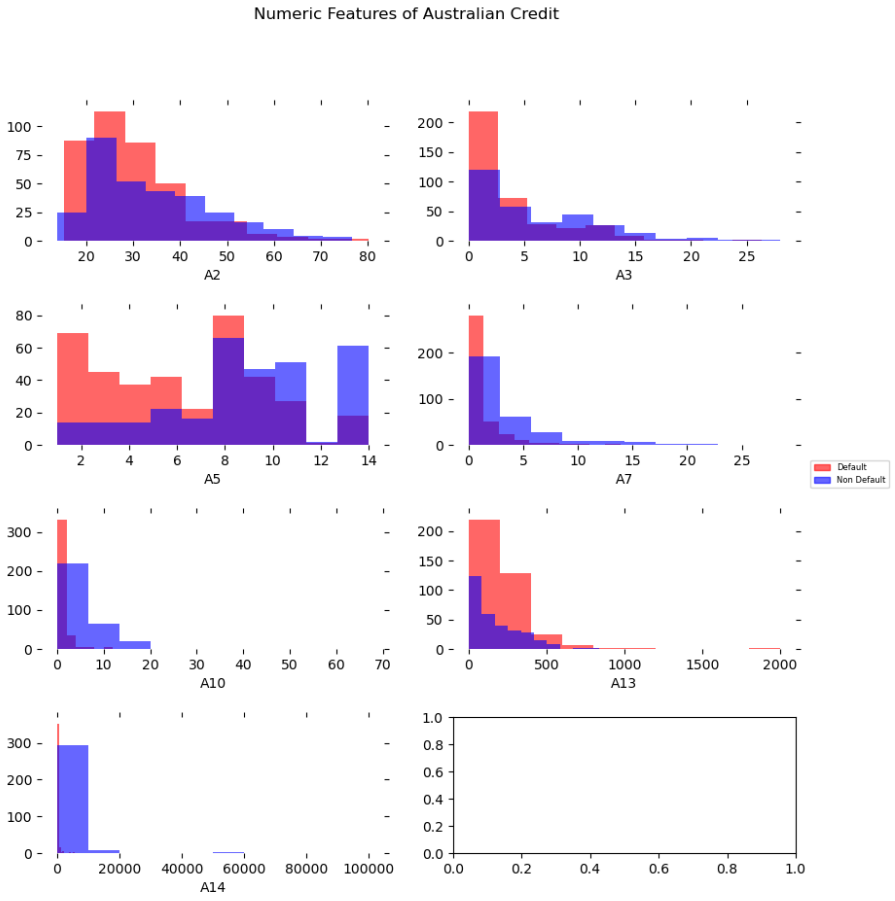


Figure B.5: GMSC numeric plot

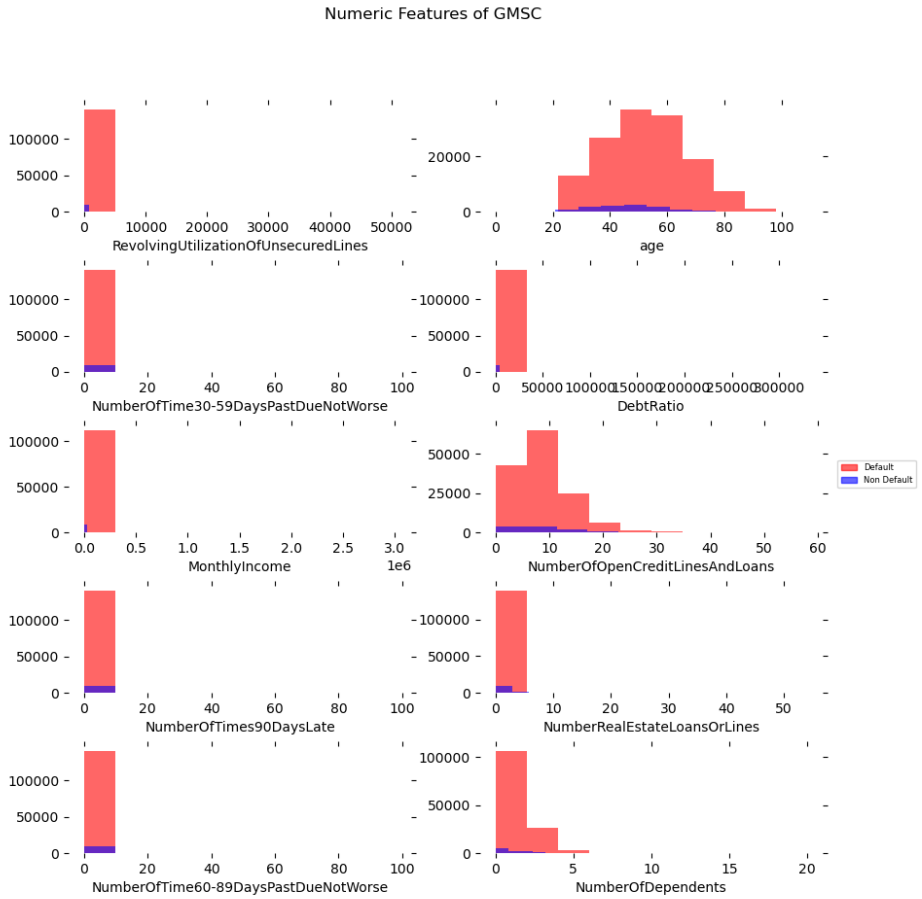


Figure B.6: GC numeric plot

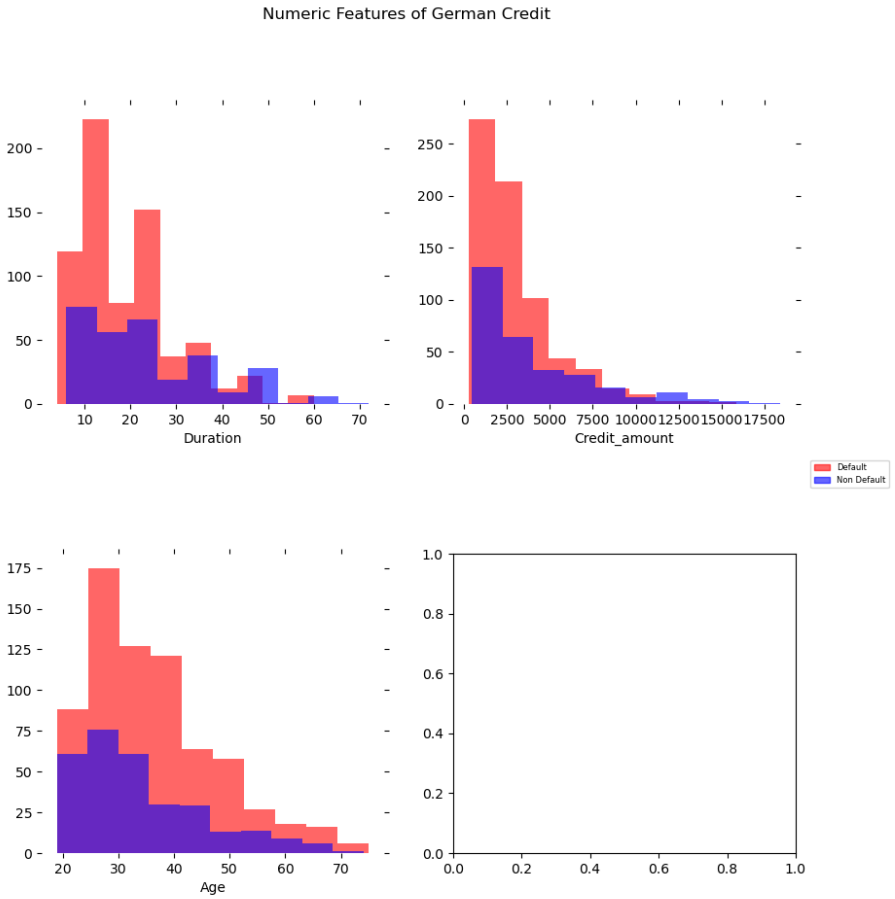


Figure B.7: TC numeric plot

B.2.1 Correlation Plots

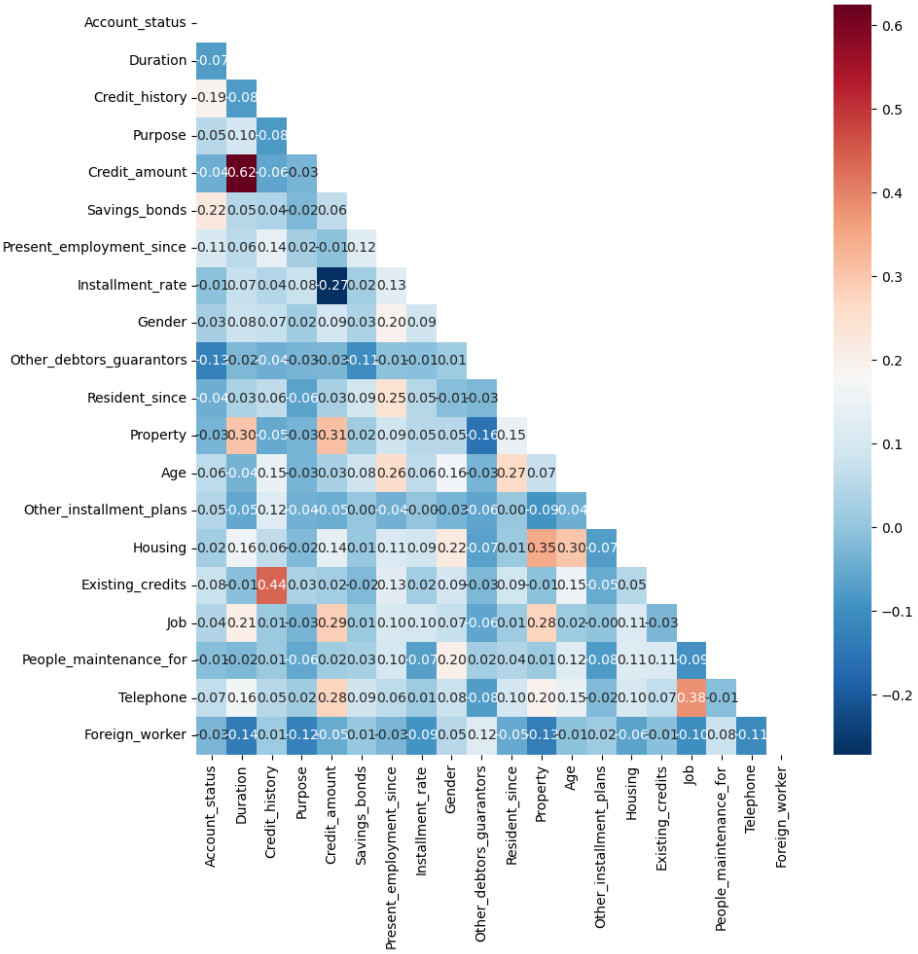


Figure B.8: GC cor plot

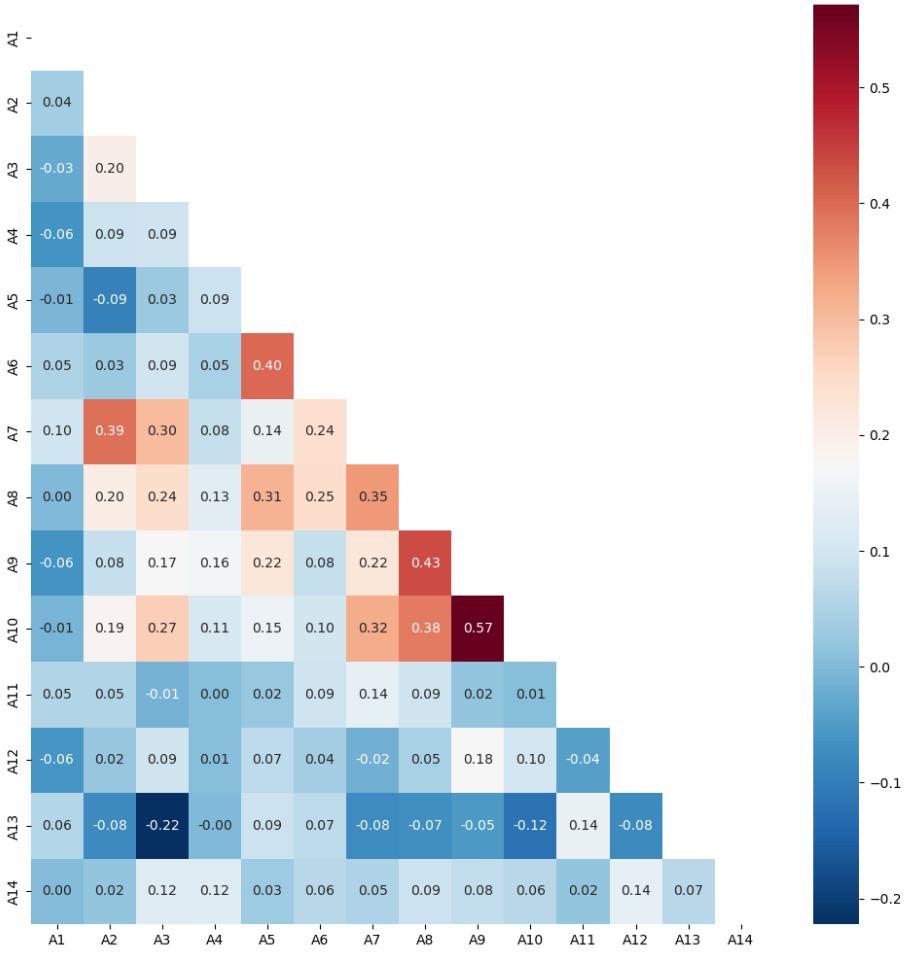


Figure B.9: AC cor plot

Appendix C

Code

```
#!/usr/bin/env python
# coding: utf-8

# In[10]:


import pandas as pd
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt
import seaborn as  sns

df1 = pd.read_csv("datasets/australian_credit.csv")
df2 = pd.read_csv("datasets/GMSC/cs-training.csv")
df3 = pd.read_csv("datasets/german_credit.csv")
df4 = pd.read_csv("datasets/UCI_Credit_Card.csv")

datasets = [df1, df2, df3, df4]
pd.set_option('display.max_columns', None)
dataset_names = ["Australian Credit", "GMSC", "German Credit", "UCI
↪  Credit Card"]
label_dict = {0: "Non Default", 1: "Default"}

# split each dataset into X and y
Xs = []
ys = []

for dataset in datasets:
Xs.append(dataset.iloc[:, :-1])
ys.append(dataset.iloc[:, -1])

ys[2] = ys[2].replace({1:0, 2:1})


# In[3]:
```

```
for i, df in enumerate(datasets):
df.describe(include='all').to_excel(dataset_names[i] + ".xlsx")
pd.isna(df).sum()


# In[4]:


# Description
for i in range(4):
display(Xs[i].describe(include='all'))


# In[9]:


# Visualizing class imbalance
for y in ys:
default_count = non_default_count = 0
for i in range(len(y)):
if y[i] == 0:
non_default_count += 1
else:
default_count += 1
print(default_count, non_default_count)
print(f"ratio: {non_default_count/default_count:.2f}")
print(f"Percentage of class 0: {non_default_count/y.shape[0]:.2f}")

# Looking for categorical and binary variables
categorical_columns = [[],[],[],[]]
binary_columns = [[],[],[],[]]
for i, X in enumerate(Xs):
for j in range(X.shape[1]):
x = X.iloc[:, j].unique()
if len(x) < 13 and len(x) > 2:
categorical_columns[i].append(j)
elif len(x) == 2:
binary_columns[i].append(j)


# In[7]:


Xs[3].columns[binary_columns[3]]


# In[13]:


for i, X in enumerate(Xs):
```

```
if i != 1:
features = X.columns
fig, axes = plt.subplots(nrows=round(len(categorical_columns[i])/2),
↪  ncols=2, figsize=(10, 10))
fig.subplots_adjust(hspace=0.5)
for col, ax in zip(categorical_columns[i], axes.flat):
for lab, color in zip(range(2), ('red', 'blue')):
sns.countplot(x=X.loc[ys[i] == lab, features[col]],
color=color,
label=f"Class {label_dict[lab]}",
alpha=0.6,
ax=ax,
)
plt.suptitle(f"{dataset_names[i]} Categorical Variables")
plt.savefig(f"plot_cat{i}.png")
plt.show()


# In[8]:


for i, X in enumerate(Xs):
if i == 3:
features = X.columns
for lab, color in zip(range(2), ('red', 'blue')):
sns.countplot(x=X.loc[ys[3] == lab, 'SEX'],
color=color,
label=f"Class {label_dict[lab]}",
alpha=0.6
)
plt.suptitle(f"{dataset_names[i]} Binary Variables")
plt.savefig(f"plot_bin{i}.png")
plt.show()
if i in [0,2]:
features = X.columns
fig, axes = plt.subplots(nrows=round(len(binary_columns[i])/2), ncols=2,
↪  figsize=(10, 10))
for col, ax in zip(binary_columns[i], axes.flat):
for lab, color in zip(range(2), ('red', 'blue')):
sns.countplot(x=X.loc[ys[i] == lab, features[col]],
color=color,
label=f"Class {label_dict[lab]}",
alpha=0.6,
ax=ax,
)
plt.suptitle(f"{dataset_names[i]} Binary Variables")
plt.savefig(f"plot_bin{i}.png")
plt.show()


# In[24]:
```

```
# Find numeric features
not_numeric_columns = [[],[],[],[]]
for i in range(4):
not_numeric_columns[i] = categorical_columns[i] + binary_columns[i]
numeric_columns_index = [[],[],[],[]]
for i, X in enumerate(Xs):
numeric_columns_index[i] = [x for x in range(X.shape[1]) if x not in
↪  not_numeric_columns[i]]


for i, X in enumerate(Xs):
ncol = len(numeric_columns_index[i])
features = X.iloc[:,numeric_columns_index[i]].columns
fig, axes = plt.subplots(nrows=round(ncol/2), ncols=2, figsize=(10, 10))
fig.subplots_adjust(hspace=2)
for ax, cnt in zip(axes.ravel(), range(ncol)):
for lab, color in zip(range(2), ('red', 'blue')):
ax.hist(X.loc[ys[i] == lab, features[cnt]],
color=color,
label=f"Class {label_dict[lab]}",
alpha=0.6
)
ax.set_xlabel(features[cnt])
# hide axis ticks
ax.tick_params(axis="both", which="both", bottom="off", top="off",
labelbottom="on", left="off", right="off",
labelleft="on")
# remove axis spines
ax.spines["top"].set_visible(False)
ax.spines["right"].set_visible(False)
ax.spines["bottom"].set_visible(False)
ax.spines["left"].set_visible(False)

red_patch = mpatches.Patch(color='red', label='Default', alpha=0.6)
blue_patch = mpatches.Patch(color='blue', label='Non Default', alpha=0.6)

fig.legend(handles=[red_patch, blue_patch], loc='center right',
↪  fancybox=True, fontsize=6)
fig.suptitle("Numeric Features of " + dataset_names[i])
plt.savefig(f"plot_num{i}.png")
plt.show()


# In[31]:


import numpy as np
def createCorrelationHeatMap(data, name):
# Correlation heatmap
```

```
corr = data.corr()
mask = np.triu(np.ones_like(corr, dtype=bool))
fig = plt.figure(figsize=(10,10))
ax = sns.heatmap(corr, annot=True, fmt='.2f', mask=mask, cmap =
↪  "RdBu_r",xticklabels=True, yticklabels=True, cbar=True)
ax.xaxis.label.set_color('black')
ax.yaxis.label.set_color('black')
ax.tick_params(axis='x', colors='black')       #set the color of xticks
ax.tick_params(axis='y', colors='black')       #set the color of yticks
fig.tight_layout()  # Improves appearance a bit.
plt.savefig(name+".png")
plt.show()


# In[ ]:


createCorrelationHeatMap(Xs[1], "gmsc_cor")


# In[140]:


createCorrelationHeatMap(Xs[1], "gmsc_cor_after")


# In[33]:


createCorrelationHeatMap(Xs[0], "ac_cor")


# In[34]:


import pandas as pd


# In[75]:


to_convert = categorical_columns[2] + binary_columns[2]
for i in to_convert:
Xs[2].iloc[:, i] = Xs[2].iloc[:, i].astype("category").cat.codes


# In[76]:


createCorrelationHeatMap(Xs[2], "gc_cor")
```

```
# In[77]:


createCorrelationHeatMap(Xs[3], "tc_cor")


# In[101]:


Xs[0].describe(include='all').to_excel("describe_ac.xlsx")
Xs[1].describe(include='all').to_excel("describe_gmsc.xlsx")
Xs[2].describe(include='all').to_excel("describe_gc.xlsx")
Xs[3].describe(include='all').to_excel("describe_tc.xlsx")


# In[111]:


Xs[1].describe(include='all')


# In[112]:


def examinePercentiles(data, percentile_threshold, step_size = 0.001):
"""
# Purpose: To examine the bucketiting of data above a threshold
↪  percentile. Used to detect outliers.
# Inputs:
#    data :  list   : Data to examine.
#    percentile_threshold:  dbl   :  Upper percentile threshold cut-off.
#    step_size  : dbl : Increments for buckets.
"""
i = percentile_threshold;
number_of_points = np.empty((0))
percentile = np.empty((0))
cut_off = np.empty((0))
number_of_points_in_current_bin = np.empty((0))
while (i<1):
i = i + step_size
if i>1:
i=1

if len(percentile)<1:
number_of_points_up_to_previous_bin = 0
else:
number_of_points_up_to_previous_bin = sum(data <=
↪  data.quantile(percentile[-1]))

number_of_points_up_to_current_bin = sum(data <= data.quantile(i))
```

```
number_of_points_in_current_bin =
↪  np.append(number_of_points_in_current_bin,
↪  number_of_points_up_to_current_bin -
↪  number_of_points_up_to_previous_bin)
number_of_points = np.append(number_of_points,
↪  number_of_points_up_to_current_bin)
cut_off = np.append(cut_off, round(data.quantile(i),2))

percentile = np.append(percentile, round(i,4))
if len(percentile)>0:
continue
#print( "cur bin:" + str(number_of_points_up_to_current_bin) +  " pre
↪  bin: " + str(number_of_points_up_to_previous_bin) + " Last
↪  percentile: " + str(percentile[-1]) + " Curr percentile: " + str(i)
↪  )
df = pd.DataFrame(data =
↪  np.column_stack((percentile,number_of_points,number_of_points_in_current_bin,
↪  cut_off)), columns = ['Percentile','# Examples to Percentile','#
↪  Examples in bin','Example Level'])
return df


# In[113]:


examinePercentiles(Xs[1].DebtRatio, 0.99, 0.001)


# In[110]:


Xs[1].loc[Xs[1]["age"] == 0].index


# In[124]:


for col in ['NumberOfTime30-59DaysPastDueNotWorse',
↪  'NumberOfTime60-89DaysPastDueNotWorse', 'NumberOfTimes90DaysLate']:
display(examinePercentiles(Xs[1][col], 0.99,  0.001))


# In[141]:


Xs[1]["NumberOfTime60-89DaysPastDueNotWorse"].value_counts().sort_index()


# In[138]:
```

```
Xs[1].loc[:,'NumberOfTime30-59DaysPastDueNotWorse'].replace({96:
↪  Xs[1]["NumberOfTime30-59DaysPastDueNotWorse"].median(),
^^I98: Xs[1]["NumberOfTime30-59DaysPastDueNotWorse"].median()},
↪  inplace=True)
Xs[1].loc[:,"NumberOfTime60-89DaysPastDueNotWorse"].replace({96:
↪  Xs[1]["NumberOfTime60-89DaysPastDueNotWorse"].median(),
^^I98: Xs[1]["NumberOfTime60-89DaysPastDueNotWorse"].median()},
↪  inplace=True)
Xs[1].loc[:,"NumberOfTimes90DaysLate"].replace({96:
↪  Xs[1]["NumberOfTimes90DaysLate"].median(),
^^I98: Xs[1]["NumberOfTimes90DaysLate"].median()}, inplace=True)


#!/usr/bin/env python
# coding: utf-8

# In[1]:


import pandas as pd
import numpy as np
# Preparation and preprocessing
from sklearn.preprocessing import StandardScaler, RobustScaler,
↪  MinMaxScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import FunctionTransformer
# Pipeline
from sklearn.pipeline import Pipeline
# Models
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC, LinearSVC

# Performance evaluation
from sklearn.model_selection import cross_val_score, StratifiedKFold,
↪  cross_validate, GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import accuracy_score, roc_curve, roc_auc_score,
↪  make_scorer
from sklearn.metrics import f1_score, precision_score, recall_score,
↪  cohen_kappa_score
from scipy.stats import ks_2samp


# In[2]:
```

```
df1 = pd.read_csv("datasets/australian_credit.csv")
df2 = pd.read_csv("datasets/GMSC/cs-training.csv")
df3 = pd.read_csv("datasets/german_credit.csv")
df4 = pd.read_csv("datasets/UCI_Credit_Card.csv")

datasets = [df1, df2, df3, df4]

# split each dataset into X and y
Xs = []
ys = []

for dataset in datasets:
Xs.append(dataset.iloc[:, :-1])
ys.append(dataset.iloc[:, -1])

ys[2] = ys[2].replace({1: 0, 2: 1})

Xs[3].iloc[:, 2] = Xs[3].iloc[:, 2].replace({1: 0, 2: 1})
to_drop = Xs[3].columns[[0, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17]]
Xs[3] = Xs[3].drop(to_drop, axis = 1)

Xs[1].loc[65695, "age"] = Xs[1]["age"].median() # Replace age 0

# Fix late payments vars
Xs[1].loc[:,'NumberOfTime30-59DaysPastDueNotWorse'] =
↪  Xs[1].loc[:,'NumberOfTime30-59DaysPastDueNotWorse'].replace(
{96: Xs[1]["NumberOfTime30-59DaysPastDueNotWorse"].median(),
^^I98: Xs[1]["NumberOfTime30-59DaysPastDueNotWorse"].median()})
Xs[1].loc[:,"NumberOfTime60-89DaysPastDueNotWorse"] =
↪  Xs[1].loc[:,"NumberOfTime60-89DaysPastDueNotWorse"].replace(
{96: Xs[1]["NumberOfTime60-89DaysPastDueNotWorse"].median(),
^^I98: Xs[1]["NumberOfTime60-89DaysPastDueNotWorse"].median()})
Xs[1].loc[:,"NumberOfTimes90DaysLate"] =
↪  Xs[1].loc[:,"NumberOfTimes90DaysLate"].replace(
{96: Xs[1]["NumberOfTimes90DaysLate"].median(),
^^I98: Xs[1]["NumberOfTimes90DaysLate"].median()})

# Looking for categorical and binary variables
categorical_columns = [[], [], [], []]
binary_columns = [[], [], [], []]
for i, X in enumerate(Xs):
for j in range(X.shape[1]):
x = X.iloc[:, j].unique()
if len(x) < 13 and len(x) > 2:
categorical_columns[i].append(j)
elif len(x) == 2:
binary_columns[i].append(j)

# Find numeric features
```

```
not_numeric_columns = [[], [], [], []]
for i in range(4):
not_numeric_columns[i] = categorical_columns[i] + binary_columns[i]
numeric_columns_index = [[], [], [], []]
for i, X in enumerate(Xs):
numeric_columns_index[i] = [x for x in range(
X.shape[1]) if x not in not_numeric_columns[i]]


Xs[2].iloc[:, binary_columns[2]] = pd.get_dummies(
Xs[2].iloc[:, binary_columns[2]], drop_first=True, dtype=int).iloc[:, [1,
↪  0, 2, 3]]

Xs[2].iloc[:, 17] = Xs[2].iloc[:, 17].replace({1: 0, 2: 1})


# In[5]:


# Transforming the data
ct0 = ColumnTransformer([
("standardised", StandardScaler(), ['A2']),
("robust", RobustScaler(), ['A3', 'A5', 'A7', 'A10', 'A13', 'A14']),
("categorical", OneHotEncoder(
handle_unknown='ignore'), ['A4', 'A6', 'A12'])
])

impute_and_scale = Pipeline([
("imputer", SimpleImputer(strategy='median')),
("scaler", RobustScaler())
])

ct1 = ColumnTransformer([
("impute_and_scale", impute_and_scale, ['MonthlyIncome',
↪  'NumberOfDependents']),
("standardised", StandardScaler(), ["age"]),
("robust", RobustScaler(), ['RevolvingUtilizationOfUnsecuredLines',
'NumberOfTime30-59DaysPastDueNotWorse', 'DebtRatio',
'NumberOfOpenCreditLinesAndLoans',
'NumberRealEstateLoansOrLines', 'NumberOfTime60-89DaysPastDueNotWorse',
↪  'NumberOfTimes90DaysLate'])
])

to_dense_transformer = FunctionTransformer(lambda x: x.toarray(),
↪  accept_sparse=True)
ct2 = ColumnTransformer([
("standardised", StandardScaler(), ["Age"]),
("robust", RobustScaler(), ['Duration', 'Credit_amount']),
("categorical", OneHotEncoder(
handle_unknown='ignore'), categorical_columns[2])
])
```

```
ct3 = ColumnTransformer([
("standardised", StandardScaler(), ["AGE"]),
("robust", RobustScaler(), ['LIMIT_BAL', 'BILL_AMT1', 'PAY_AMT1',
'PAY_AMT2', 'PAY_AMT3', 'PAY_AMT4', 'PAY_AMT5',
'PAY_AMT6']),
("categorical", OneHotEncoder(
handle_unknown='ignore'), ['EDUCATION', 'MARRIAGE', 'PAY_0'])
])


# For mlp do a random grid search

# In[22]:


skf_inner = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
skf_outer = StratifiedKFold(n_splits=10, shuffle=True, random_state=43)


# In[23]:


# MLP tuning
param_grid = {
^^I'classifier__hidden_layer_sizes': [(100,), (200,), (200, 100, ), (100,
↪  50, )],
^^I'classifier__activation': ['logistic', 'tanh', 'relu'],
^^I'classifier__max_iter': [2000, 2500]
}


# In[24]:


Pipeline1_mlp = Pipeline(steps=[("preprocessor", ct1), ("classifier",
MLPClassifier(random_state=7, early_stopping=True))])
gmsc_mlp_hp = RandomizedSearchCV(Pipeline1_mlp, param_grid, cv=skf_inner,
↪  scoring="f1", n_jobs=-1).fit(Xs[1], ys[1])
print(gmsc_mlp_hp.best_estimator_)


# In[36]:


ac_mlp_hp = GridSearchCV(Pipeline0_mlp, param_grid, cv=skf_inner,
↪  scoring="f1", n_jobs=-1).fit(Xs[0], ys[0])
print(ac_mlp_hp.best_estimator_)


# In[116]:
```

```
Pipeline2_mlp = Pipeline(steps=[("preprocessor", ct2), ("classifier",
MLPClassifier(random_state=7, early_stopping=True))])
gc_mlp_hp = GridSearchCV(Pipeline2_mlp, param_grid, cv=skf_inner,
↪  scoring="f1", n_jobs=-1).fit(Xs[2], ys[2])
print(gc_mlp_hp.best_estimator_)


# In[52]:


param_grid = {
^^I'classifier__hidden_layer_sizes': [(100,), (200,), (200, 100, ), (100,
↪  50, )],
^^I'classifier__activation': ['logistic', 'tanh', 'relu'],
^^I'classifier__max_iter': [2000, 2500]
}
Pipeline3_mlp = Pipeline(steps=[("preprocessor", ct3), ("classifier",
MLPClassifier(random_state=7, early_stopping=True))])
tc_mlp_hp = RandomizedSearchCV(Pipeline3_mlp, param_grid, cv=skf_inner,
↪  scoring="f1", n_jobs=-1).fit(Xs[3], ys[3])
print(tc_mlp_hp.best_estimator_)
Pipeline3_mlp = tc_mlp_hp.best_estimator_


# In[ ]:


Pipeline0_mlp = Pipeline(steps=[("preprocessor", ct0), ("classifier",
MLPClassifier(early_stopping=True, hidden_layer_sizes=(200, 100),
max_iter=2000,random_state=7))])
Pipeline1_mlp = Pipeline(steps=[("preprocessor", ct1), ("classifier",
MLPClassifier(activation='logistic', early_stopping=True,
↪  hidden_layer_sizes=(200,100),
max_iter=2500, random_state=7))])
Pipeline2_mlp = Pipeline(steps=[("preprocessor", ct2), ("classifier",
MLPClassifier(activation='tanh', hidden_layer_sizes=(200, 100),
max_iter=2000, random_state=7,
solver='lbfgs'))])
Pipeline3_mlp = Pipeline(steps=[("preprocessor", ct3), ("classifier",
MLPClassifier(early_stopping=True,
hidden_layer_sizes=(200, 100), max_iter=2500,
random_state=7))])


# In[70]:


param_grid = {
^^I'classifier__C': [1, 5, 10],
```

```
^^I'classifier__solver': ['liblinear', 'newton-cg']
}
Pipeline1_lr = Pipeline(steps=[("preprocessor", ct1), ("classifier",
LogisticRegression(max_iter=500, random_state=7))])
grid_search = GridSearchCV(Pipeline1_lr, param_grid, scoring="f1",
↪  cv=skf_inner, n_jobs=-1)
best_model = grid_search.fit(Xs[1], ys[1]).best_estimator_


# In[71]:


print(best_model)


# In[75]:


# Logistic Regression tuning
param_grid = {
^^I'classifier__C': [1, 5, 10],
^^I'classifier__solver': ['liblinear', 'newton-cg']
}

lr_pipelines = [Pipeline0_lr, Pipeline1_lr, Pipeline2_lr, Pipeline3_lr]
for i, X in enumerate(Xs):
pipeline = lr_pipelines[i]

# Perform grid search
grid_search = GridSearchCV(pipeline, param_grid, scoring="f1",
↪  cv=skf_inner, n_jobs=-1)
best_model = grid_search.fit(X, ys[i]).best_estimator_

# Print the best estimator for the current dataset
print(f"Best model for dataset {i}: {best_model}")


# In[47]:


# Logistic Regression tuning
param_grid = {
^^I'classifier__C': [0.1, 1, 5, 10],
^^I'classifier__solver': ['lbfgs', 'liblinear', 'newton-cg']
}
Pipeline3_lr = Pipeline(steps=[("preprocessor", ct3), ("classifier",
LogisticRegression(max_iter=500, random_state=7))])
best_estim = GridSearchCV(Pipeline3_lr, param_grid, scoring="f1",
↪  cv=skf_inner, n_jobs=-1).fit(Xs[3], ys[3]).best_estimator_
print(best_estim)
Pipeline3_lr = best_estim
```

```
# In[ ]:


Pipeline0_lr = Pipeline(steps=[("preprocessor", ct0), ("classifier",
LogisticRegression(C=1, solver='liblinear'))])
Pipeline1_lr = Pipeline(steps=[("preprocessor", ct1), ("classifier",
LogisticRegression(C=5, solver='newton-cg'))])
Pipeline2_lr = Pipeline(steps=[("preprocessor", ct2), ("classifier",
LogisticRegression(C=1, solver='newton-cg'))])
Pipeline3_lr = Pipeline(steps=[("preprocessor", ct3), ("classifier",
LogisticRegression(C=10, max_iter=500, random_state=7))])


# In[50]:


param_grid = {
^^I'classifier__max_depth': [None, 10, 20, 30],
^^I'classifier__min_samples_split': [2, 5, 10],
^^I'classifier__min_samples_leaf': [1, 2, 4],
^^I'classifier__max_features': ['sqrt', 'log2'],
}
Pipeline3_rf =  Pipeline(steps=[("preprocessor", ct3), ("classifier",
RandomForestClassifier(random_state=7))])
random_search = RandomizedSearchCV(Pipeline3_rf, param_grid,
↪  scoring="f1", cv=skf_inner, verbose=1, n_jobs=-1)
best_estim = random_search.fit(Xs[3], ys[3]).best_estimator_
print(best_estim)
Pipeline3_rf = best_estim


# In[83]:


# Random Forest tuning

param_grid = {
^^I'classifier__max_depth': [None, 10, 20, 30],
^^I'classifier__min_samples_split': [2, 5, 10],
^^I'classifier__min_samples_leaf': [1, 2, 4],
^^I'classifier__max_features': ['sqrt', 'log2'],
}

rf_pipelines = [Pipeline0_rf, Pipeline1_rf, Pipeline2_rf, Pipeline3_rf]

for i, X in enumerate(Xs):
pipeline = rf_pipelines[i]

# Perform grid search
```

```
random_search = RandomizedSearchCV(pipeline, param_grid, scoring="f1",
↪  cv=skf_inner, verbose=1, n_jobs=-1)
best_model = random_search.fit(X, ys[i]).best_estimator_

# Print the best estimator for the current dataset
print(f"Best model for dataset {i}: {best_model}")


# In[ ]:


Pipeline0_rf = Pipeline(steps=[("preprocessor", ct0), ("classifier",
RandomForestClassifier(max_depth=30,
↪  max_features='log2',min_samples_split=5))])
Pipeline1_rf = Pipeline(steps=[("preprocessor", ct1), ("classifier",
RandomForestClassifier(max_features='log2', min_samples_split=10))])
Pipeline2_rf = Pipeline(steps=[("preprocessor", ct2), ("classifier",
RandomForestClassifier(max_depth=30,
↪  min_samples_leaf=2,min_samples_split=10))])
Pipeline3_rf = Pipeline(steps=[("preprocessor", ct3), ("classifier",
RandomForestClassifier(max_depth=30, min_samples_split=10,
random_state=7))])


# In[93]:


# knn tuning

param_grid = {
^^I'classifier__n_neighbors': [3, 5, 7, 10, 15],
^^I'classifier__weights': ['uniform', 'distance'],
^^I'classifier__leaf_size': [20, 30, 40],
}

knn_pipelines = [Pipeline0_knn, Pipeline1_knn, Pipeline2_knn,
↪  Pipeline3_knn]

for i, X in enumerate(Xs):
pipeline = knn_pipelines[i]

# Perform grid search
random_search = RandomizedSearchCV(pipeline, param_grid, scoring="f1",
↪  cv=skf_inner, verbose=1, n_jobs=-1)
best_model = random_search.fit(X, ys[i]).best_estimator_

# Print the best estimator for the current dataset
print(f"Best model for dataset {i}: {best_model}")


# In[54]:
```

```
param_grid = {
^^I'classifier__n_neighbors': [3, 5, 7, 10, 15],
^^I'classifier__weights': ['uniform', 'distance'],
^^I'classifier__leaf_size': [20, 30, 40],
}
Pipeline3_knn = Pipeline(steps=[("preprocessor", ct3), ("classifier",
KNeighborsClassifier(algorithm='ball_tree'))])

random_search = RandomizedSearchCV(Pipeline3_knn, param_grid,
↪  scoring="f1", cv=skf_inner, verbose=3, n_jobs=-1)
best_estim = random_search.fit(Xs[3], ys[3]).best_estimator_
print(best_estim)
Pipeline3_knn = best_estim


# In[ ]:


Pipeline0_knn = Pipeline(steps=[("preprocessor", ct0), ("classifier",
KNeighborsClassifier(leaf_size=40, n_neighbors=15, weights='distance'))])
Pipeline1_knn = Pipeline(steps=[("preprocessor", ct1), ("classifier",
KNeighborsClassifier(leaf_size=40, n_neighbors=3))])
Pipeline2_knn = Pipeline(steps=[("preprocessor", ct2), ("classifier",
KNeighborsClassifier(leaf_size=20, n_neighbors=3, weights='distance'))])
Pipeline3_knn = Pipeline(steps=[("preprocessor", ct3), ("classifier",
KNeighborsClassifier(algorithm='ball_tree',
leaf_size=40, n_neighbors=10, weights='distance'))])


# In[61]:


# dt tuning

param_grid = {
^^I'classifier__max_depth': [None, 5, 10, 15],
^^I'classifier__min_samples_leaf': [1, 2, 4, 6],
^^I'classifier__max_features': ['sqrt', 'log2']
}
Pipeline0_dt = Pipeline(steps=[("preprocessor", ct0), ("classifier",
DecisionTreeClassifier(random_state=7))])
Pipeline1_dt = Pipeline(steps=[("preprocessor", ct1), ("classifier",
DecisionTreeClassifier(random_state=7))])
Pipeline2_dt = Pipeline(steps=[("preprocessor", ct2), ("classifier",
DecisionTreeClassifier(random_state=7))])
Pipeline3_dt = Pipeline(steps=[("preprocessor", ct3), ("classifier",
DecisionTreeClassifier(random_state=7))])

dt_pipelines = [Pipeline0_dt, Pipeline1_dt, Pipeline2_dt, Pipeline3_dt]
```

```
for i in [0,2,3,1]:
grid_search = GridSearchCV(dt_pipelines[i], param_grid, scoring="f1",
↪  cv=skf_inner, verbose=1, n_jobs=-1)
best_estim = grid_search.fit(Xs[i], ys[i]).best_estimator_
print(best_estim)
dt_pipelines[i] = best_estim
Pipeline0_dt, Pipeline1_dt, Pipeline2_dt, Pipeline3_dt = dt_pipelines


# In[ ]:


Pipeline0_dt = Pipeline(steps=[("preprocessor", ct0), ("classifier",
DecisionTreeClassifier(max_depth=10, max_features='sqrt',
min_samples_leaf=6, random_state=7))])
Pipeline1_dt = Pipeline(steps=[("preprocessor", ct1), ("classifier",
DecisionTreeClassifier(max_features='sqrt',
min_samples_leaf=6, random_state=7))])
Pipeline2_dt = Pipeline(steps=[("preprocessor", ct2), ("classifier",
DecisionTreeClassifier(max_depth=10, max_features='sqrt',
min_samples_leaf=6, random_state=7))])
Pipeline3_dt = Pipeline(steps=[("preprocessor", ct3), ("classifier",
DecisionTreeClassifier(max_depth=10, max_features='sqrt',
random_state=7))])


# In[60]:


# svm tuning

param_grid = {
^^I'classifier__C': [0.1, 1, 10, 100] # Reg param
}
Pipeline0_svm = Pipeline(steps=[("preprocessor", ct0), ("classifier",
SVC(random_state=7, max_iter=10000, probability=True))])
Pipeline1_svm = Pipeline(steps=[("preprocessor", ct1), ("classifier",
LinearSVC(random_state=7, max_iter=10000))])
Pipeline2_svm = Pipeline(steps=[("preprocessor", ct2), ("classifier",
SVC(random_state=7, max_iter=10000, probability=True))])
Pipeline3_svm = Pipeline(steps=[("preprocessor", ct3), ("classifier",
LinearSVC(random_state=7, max_iter=10000))])

svm_pipelines = [Pipeline0_svm, Pipeline1_svm, Pipeline2_svm,
↪  Pipeline3_svm]

for i in [0,2,3,1]:
grid_search = GridSearchCV(svm_pipelines[i], param_grid, scoring="f1",
↪  cv=skf_inner, verbose=1, n_jobs=-1)
best_estim = grid_search.fit(Xs[i], ys[i]).best_estimator_
```

```
print(best_estim)
svm_pipelines[i] = best_estim
Pipeline0_svm, Pipeline1_svm, Pipeline2_svm, Pipeline3_svm =
↪  svm_pipelines


# In[67]:


# SVM pipelines
Pipeline0_svm = Pipeline(steps=[("preprocessor", ct0), ("classifier",
SVC(C=10, random_state=7, max_iter=10000, probability=True))])
Pipeline1_svm = Pipeline(steps=[("preprocessor", ct1), ("classifier",
LinearSVC(C=0.1, random_state=7, max_iter=10000))])
Pipeline2_svm = Pipeline(steps=[("preprocessor", ct2), ("classifier",
SVC(C=10, random_state=7, max_iter=10000, probability=True))])
Pipeline3_svm = Pipeline(steps=[("preprocessor", ct3), ("classifier",
LinearSVC(C=0.1, random_state=7, max_iter=10000))])


#!/usr/bin/env python
# coding: utf-8

# In[14]:


import pandas as pd
import numpy as np
# Preparation and preprocessing
from sklearn.preprocessing import StandardScaler, RobustScaler,
↪  MinMaxScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import FunctionTransformer
# Pipeline
from sklearn.pipeline import Pipeline
# Models
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC, LinearSVC

# Performance evaluation
from sklearn.model_selection import cross_val_score, StratifiedKFold,
↪  cross_validate, GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
```

```
from sklearn.metrics import accuracy_score, roc_curve, roc_auc_score,
↪  make_scorer
from sklearn.metrics import f1_score, precision_score, recall_score,
↪  cohen_kappa_score
from scipy.stats import ks_2samp


# In[18]:


df1 = pd.read_csv("datasets/australian_credit.csv")
df2 = pd.read_csv("datasets/GMSC/cs-training.csv")
df3 = pd.read_csv("datasets/german_credit.csv")
df4 = pd.read_csv("datasets/UCI_Credit_Card.csv")

datasets = [df1, df2, df3, df4]

# split each dataset into X and y
Xs = []
ys = []

for dataset in datasets:
Xs.append(dataset.iloc[:, :-1])
ys.append(dataset.iloc[:, -1])

ys[2] = ys[2].replace({1: 0, 2: 1})

Xs[3].iloc[:, 2] = Xs[3].iloc[:, 2].replace({1: 0, 2: 1})
to_drop = Xs[3].columns[[0, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17]]
Xs[3] = Xs[3].drop(to_drop, axis = 1)

Xs[1].loc[65695, "age"] = Xs[1]["age"].median() # Replace age 0

# Fix late payments vars
Xs[1].loc[:,'NumberOfTime30-59DaysPastDueNotWorse'] =
↪  Xs[1].loc[:,'NumberOfTime30-59DaysPastDueNotWorse'].replace(
{96: Xs[1]["NumberOfTime30-59DaysPastDueNotWorse"].median(),
^^I98: Xs[1]["NumberOfTime30-59DaysPastDueNotWorse"].median()})
Xs[1].loc[:,"NumberOfTime60-89DaysPastDueNotWorse"] =
↪  Xs[1].loc[:,"NumberOfTime60-89DaysPastDueNotWorse"].replace(
{96: Xs[1]["NumberOfTime60-89DaysPastDueNotWorse"].median(),
^^I98: Xs[1]["NumberOfTime60-89DaysPastDueNotWorse"].median()})
Xs[1].loc[:,"NumberOfTimes90DaysLate"] =
↪  Xs[1].loc[:,"NumberOfTimes90DaysLate"].replace(
{96: Xs[1]["NumberOfTimes90DaysLate"].median(),
^^I98: Xs[1]["NumberOfTimes90DaysLate"].median()})

# Looking for categorical and binary variables
categorical_columns = [[], [], [], []]
binary_columns = [[], [], [], []]
for i, X in enumerate(Xs):
```

```
for j in range(X.shape[1]):
x = X.iloc[:, j].unique()
if len(x) < 13 and len(x) > 2:
categorical_columns[i].append(j)
elif len(x) == 2:
binary_columns[i].append(j)

# Find numeric features
not_numeric_columns = [[], [], [], []]
for i in range(4):
not_numeric_columns[i] = categorical_columns[i] + binary_columns[i]
numeric_columns_index = [[], [], [], []]
for i, X in enumerate(Xs):
numeric_columns_index[i] = [x for x in range(
X.shape[1]) if x not in not_numeric_columns[i]]


Xs[2].iloc[:, binary_columns[2]] = pd.get_dummies(
Xs[2].iloc[:, binary_columns[2]], drop_first=True, dtype=int).iloc[:, [1,
↪  0, 2, 3]]

Xs[2].iloc[:, 17] = Xs[2].iloc[:, 17].replace({1: 0, 2: 1})


# In[19]:


# Transforming the data
ct0 = ColumnTransformer([
("standardised", StandardScaler(), ['A2']),
("robust", RobustScaler(), ['A3', 'A5', 'A7', 'A10', 'A13', 'A14']),
("categorical", OneHotEncoder(
handle_unknown='ignore'), ['A4', 'A6', 'A12'])
])

impute_and_scale = Pipeline([
("imputer", SimpleImputer(strategy='median')),
("scaler", RobustScaler())
])

ct1 = ColumnTransformer([
("impute_and_scale", impute_and_scale, ['MonthlyIncome',
↪  'NumberOfDependents']),
("standardised", StandardScaler(), ["age"]),
("robust", RobustScaler(), ['RevolvingUtilizationOfUnsecuredLines',
'NumberOfTime30-59DaysPastDueNotWorse', 'DebtRatio',
'NumberOfOpenCreditLinesAndLoans',
'NumberRealEstateLoansOrLines', 'NumberOfTime60-89DaysPastDueNotWorse',
↪  'NumberOfTimes90DaysLate'])
])
```

```
to_dense_transformer = FunctionTransformer(lambda x: x.toarray(),
↪  accept_sparse=True)
ct2 = ColumnTransformer([
("standardised", StandardScaler(), ["Age"]),
("robust", RobustScaler(), ['Duration', 'Credit_amount']),
("categorical", OneHotEncoder(
handle_unknown='ignore'), categorical_columns[2])
])

ct3 = ColumnTransformer([
("standardised", StandardScaler(), ["AGE"]),
("robust", RobustScaler(), ['LIMIT_BAL', 'BILL_AMT1', 'PAY_AMT1',
'PAY_AMT2', 'PAY_AMT3', 'PAY_AMT4', 'PAY_AMT5',
'PAY_AMT6']),
("categorical", OneHotEncoder(
handle_unknown='ignore'), ['EDUCATION', 'MARRIAGE', 'PAY_0'])
])


# In[17]:


skf_inner = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
skf_outer = StratifiedKFold(n_splits=10, shuffle=True, random_state=43)


# In[29]:


Pipeline0_lr = Pipeline(steps=[("preprocessor", ct0), ("classifier",
LogisticRegression(C=1, solver='liblinear'))])
Pipeline1_lr = Pipeline(steps=[("preprocessor", ct1), ("classifier",
LogisticRegression(C=5, max_iter=500, random_state=7,
solver='newton-cg'))])
Pipeline2_lr = Pipeline(steps=[("preprocessor", ct2), ("classifier",
LogisticRegression(C=1, solver='newton-cg'))])
Pipeline3_lr = Pipeline(steps=[("preprocessor", ct3), ("classifier",
LogisticRegression(C=10, max_iter=500, random_state=7))])


# In[47]:


Pipeline0_mlp = Pipeline(steps=[("preprocessor", ct0), ("classifier",
MLPClassifier(early_stopping=True, hidden_layer_sizes=(200, 100),
max_iter=2000,random_state=7))])
Pipeline1_mlp = Pipeline(steps=[("preprocessor", ct1), ("classifier",
MLPClassifier(activation='logistic', early_stopping=True,
↪  hidden_layer_sizes=(200,),
max_iter=2500, random_state=7))])
Pipeline2_mlp = Pipeline(steps=[("preprocessor", ct2), ("classifier",
```

```
MLPClassifier(activation='tanh', hidden_layer_sizes=(200, 100),
max_iter=2000, random_state=7,
solver='lbfgs'))])
Pipeline3_mlp = Pipeline(steps=[("preprocessor", ct3), ("classifier",
MLPClassifier(early_stopping=True,
hidden_layer_sizes=(200, 100), max_iter=2500,
random_state=7))])
```

```
# In[30]:
```

```
Pipeline0_rf = Pipeline(steps=[("preprocessor", ct0), ("classifier",
RandomForestClassifier(max_depth=30,
↪  max_features='log2',min_samples_split=5))])
Pipeline1_rf = Pipeline(steps=[("preprocessor", ct1), ("classifier",
RandomForestClassifier(max_features='log2', min_samples_split=10))])
Pipeline2_rf = Pipeline(steps=[("preprocessor", ct2), ('to_dense',
↪  to_dense_transformer), ("classifier",
RandomForestClassifier(max_depth=30,
↪  min_samples_leaf=2,min_samples_split=10))])
Pipeline3_rf = Pipeline(steps=[("preprocessor", ct3), ("classifier",
RandomForestClassifier(max_depth=30, min_samples_split=10,
random_state=7))])
```

```
# In[31]:
```

```
Pipeline0_knn = Pipeline(steps=[("preprocessor", ct0), ("classifier",
KNeighborsClassifier(leaf_size=40, n_neighbors=15, weights='distance'))])
Pipeline1_knn = Pipeline(steps=[("preprocessor", ct1), ("classifier",
KNeighborsClassifier(leaf_size=40, n_neighbors=3))])
Pipeline2_knn = Pipeline(steps=[("preprocessor", ct2), ('to_dense',
↪  to_dense_transformer), ("classifier",
KNeighborsClassifier(leaf_size=20, n_neighbors=3, weights='distance'))])
Pipeline3_knn = Pipeline(steps=[("preprocessor", ct3), ("classifier",
KNeighborsClassifier(algorithm='ball_tree',
leaf_size=40, n_neighbors=10, weights='distance'))])
```

```
# In[32]:
```

```
Pipeline0_dt = Pipeline(steps=[("preprocessor", ct0), ("classifier",
DecisionTreeClassifier(max_depth=10, max_features='sqrt',
min_samples_leaf=6, random_state=7))])
Pipeline1_dt = Pipeline(steps=[("preprocessor", ct1), ("classifier",
DecisionTreeClassifier(max_features='sqrt',
min_samples_leaf=6, random_state=7))])
```

```
Pipeline2_dt = Pipeline(steps=[("preprocessor", ct2), ('to_dense',
↪  to_dense_transformer), ("classifier",
DecisionTreeClassifier(max_depth=10, max_features='sqrt',
min_samples_leaf=6, random_state=7))])
Pipeline3_dt = Pipeline(steps=[("preprocessor", ct3), ("classifier",
DecisionTreeClassifier(max_depth=10, max_features='sqrt',
random_state=7))])


# In[33]:


# SVM pipelines
Pipeline0_svm = Pipeline(steps=[("preprocessor", ct0), ("classifier",
SVC(C=10, random_state=7, max_iter=10000, probability=True))])
Pipeline1_svm = Pipeline(steps=[("preprocessor", ct1), ("classifier",
LinearSVC(C=0.1, random_state=7, max_iter=10000))])
Pipeline2_svm = Pipeline(steps=[("preprocessor", ct2), ("classifier",
SVC(C=10, random_state=7, max_iter=10000, probability=True))])
Pipeline3_svm = Pipeline(steps=[("preprocessor", ct3), ("classifier",
LinearSVC(C=0.1, random_state=7, max_iter=10000))])


# In[34]:


# lda pipelines
Pipeline0_lda = Pipeline(steps=[("preprocessor", ct0), ("classifier",
LinearDiscriminantAnalysis())])
Pipeline1_lda = Pipeline(steps=[("preprocessor", ct1), ("classifier",
LinearDiscriminantAnalysis())])
Pipeline2_lda = Pipeline(steps=[("preprocessor", ct2), ('to_dense',
↪  to_dense_transformer), ("classifier",
LinearDiscriminantAnalysis())])
Pipeline3_lda = Pipeline(steps=[("preprocessor", ct3), ("classifier",
LinearDiscriminantAnalysis())])


# In[62]:


# Final training and tests

def ks_statistic(y_true, y_pred_prob):
y_true = y_true.astype(int)
return ks_2samp(y_pred_prob[y_true == 1], y_pred_prob[y_true ==
↪  0]).statistic


ks_scorer = make_scorer(ks_statistic, response_method='predict_proba')
```

```
metrics = {'accuracy': 'accuracy', 'precision': 'precision',
^^I'recall': 'recall', 'ks': ks_scorer, "kappa":
 ↪  make_scorer(cohen_kappa_score), 'roc_auc': 'roc_auc'}

def cv_pipelines(pipelines, X, y, metrics, cv):
results = []
result_means = []
for pipeline in pipelines:
result = cross_validate(
pipeline, X, y, scoring=metrics, cv=cv, return_train_score=True,
 ↪  n_jobs=-1)
results.append(pd.DataFrame(result))
result_means.append(np.mean(pd.DataFrame(result), axis=0))
return (results, result_means)

pipelines = {
^^I0 : [Pipeline0_lr, Pipeline0_rf, Pipeline0_lda, Pipeline0_knn,
^^IPipeline0_mlp, Pipeline0_dt, Pipeline0_svm],
^^I1 : [Pipeline1_lr, Pipeline1_rf, Pipeline1_lda, Pipeline1_knn,
^^IPipeline1_mlp, Pipeline1_dt],
^^I2 : [Pipeline2_lr, Pipeline2_rf, Pipeline2_lda, Pipeline2_knn,
^^IPipeline2_mlp, Pipeline2_dt, Pipeline2_svm],
^^I3 : [Pipeline3_lr, Pipeline3_rf, Pipeline3_lda, Pipeline3_knn,
^^IPipeline3_mlp, Pipeline3_dt]
}

result0, result_mean0 = cv_pipelines(pipelines[0], Xs[0], ys[0], metrics,
 ↪  cv=skf_outer)
result1, result_mean1 = cv_pipelines(pipelines[1], Xs[1], ys[1], metrics,
 ↪  cv=skf_outer)


# In[55]:

result2, result_mean2 = cv_pipelines(pipelines[2], Xs[2], ys[2], metrics,
 ↪  cv=skf_outer)
result3, result_mean3 = cv_pipelines(pipelines[3], Xs[3], ys[3], metrics,
 ↪  cv=skf_outer)

pd.concat([pd.DataFrame(result_mean0),
pd.DataFrame(result_mean1),
pd.DataFrame(result_mean2),
pd.DataFrame(result_mean3)]).to_csv('consolidated2.csv')


# In[73]:


def ks_statistic(y_true, y_pred_prob):
y_true = y_true.astype(int)
```

```
return ks_2samp(y_pred_prob[y_true == 1], y_pred_prob[y_true ==
↪  0]).statistic


ks_scorer = make_scorer(ks_statistic,
↪  response_method='decision_function')

metrics = {'accuracy': 'accuracy', 'precision': 'precision',
^^I'recall': 'recall', 'ks': ks_scorer, "kappa":
↪  make_scorer(cohen_kappa_score), 'roc_auc': 'roc_auc'}

svm1_res = cross_validate(Pipeline1_svm, Xs[1], ys[1], scoring=metrics,
↪  cv=skf_outer, return_train_score=True, n_jobs=-1)
#svm2_res = cross_validate(Pipeline3_svm, Xs[3], ys[3], scoring=metrics,
↪  cv=skf_outer, return_train_score=True, n_jobs=-1)


# In[ ]:


pd.concat([pd.DataFrame(svm1_res).mean(axis=0),
↪  pd.DataFrame(svm2_res).mean(axis=0)]).to_excel("svm.xlsx")


# In[71]:


pd.DataFrame(pd.DataFrame(mlp1).mean()).transpose().to_clipboard()


# In[75]:


pd.DataFrame(svm1_res).mean().to_clipboard()


# In[4]:


from scipy.stats import friedmanchisquare
ac_ranks = [3.55,1.45,5.55,5.45,2.7,6.7,2.6]
gmsc_ranks = [5.4,2.4,4.2,4.85,3.1,2.95,3.85]
gc_ranks = [2.7,2.8,2.9,5.45,4.35,7,2.8]
tc_ranks = [3.2,2.05,3.6,6.7,3.25,6.3,2.9]

res = friedmanchisquare(ac_ranks, gmsc_ranks, gc_ranks, tc_ranks)
if res.pvalue < 0.05:
print(f"The X2 statistic is {res.statistic:.2f} with a p value =
↪  {res.pvalue:.2f}, we reject H0")
else:
```

```
print(f"The X2 statistic is {res.statistic:.2f} with a p value =
↪  {res.pvalue:.2f}, we fail to reject H0")
```