Loan Credit Scoring: Probabilistic Classification of Loan Defaults

15 March 2021

Abstract

In this project, proof-of-concept predictive loan default probability-based scoring models for no/limited credit history applicantions were developed. Through the models, the project also hopes to identify key/important consumer features that contribute to the loan risk scores. Binary probabilistic classification models were built and several techniques such as cross-validation, hyperparameter tuning, probability calibration and under/oversampling were carried out to develop and improve the models. Overall, the final model built upon XGBoost's implementation of a gradient boosted tree was evaluated for its probability prediction performance on the testing dataset where it performed 46.9%, 15.6% and 14.5% better in the ROC AUC, log loss and Brier score metrics than naïve classifiers. In terms of classification performance, it performed 120% better than the naïve classifier based on F1 scores when the model was G-Means (sensitivity/specificity) optimised. Finally, feature importances were explored, highlighting country and language as key features.

Contents

1 Introduction

Traditionally, information from external credit bureau and extensive past payment behaviour is used heavily to develop credit scoring models which support the credit decisioning process of the lending function in banks. Alongside busines rules developed within the banks, the credit scores are then used to approve or deny credit applications [1].

The heavy reliance of traditional credit scoring models on credit bureau information creates challenges for a huge proportion of consumers, who have little to no relevant credit file such as recent graduates, pensioners, retirees etc., to gain access to loan and credit facilities. This leaves large segments of the population who are underbanked or unbanked, some of which are potential revenue sources for the bank.

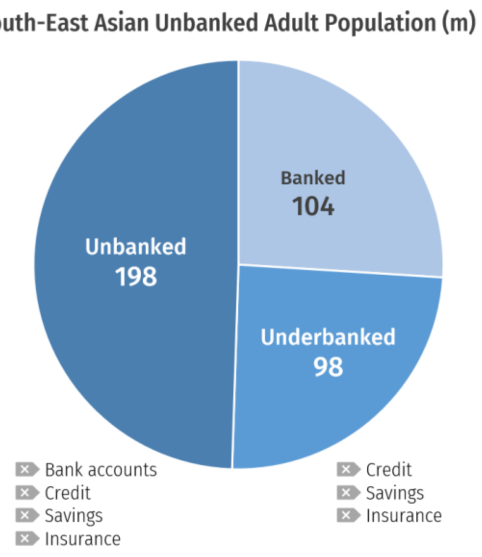


Figure 1 – Large proportions of South-East Asian adult population remains unbanked and underbanked, with limited access to various financial facilities [2].

As the credit scoring process is generally a data-intensive one, it is an area of interest where data analytics and machine learning models can be employed to possibly produce better, quicker and informed approve or deny credit application decisions.

1.1 Objectives

1.1.1 Business Objectives

In particular, this project hopes to focus on the underbanked segments, consumers who are "thin-file" or "no-files", with very limited to no credit history who may be better served by such analytical models.

The project hopes to develop a credible credit risk scoring model for unsecured retail loan applicants who have limited credit history with the bank. The credit risk score developed would be based on the probability of a default (as defined in Section 2.1) on the loan by the consumer, the higher the probability of loan default, the higher the credit risk.

The project also hopes to identify key consumer information/features which have the greatest impact on the decisioning, which can be used to develop a credit scorecard.

As the decision produced by such an analytical model have a huge effect on the bank's operations and on the financial inclusion of different consumer segments, there are operational and ethical considerations which warrant a deeper, separate study. As such, the project would be a proof-of-concept (POC) at this stage.

1.1.2 Data Analytics Objectives

The project thus aims to develop a POC, predictive loan default probability-based scoring model for no/limited credit history applicants, and identify key/important consumer features that contribute to the loan risk scores.

1.2 Limitations

Due to privacy policies, bank's production data was not used. Instead, generic developmental datasets (hereafter referred to as "Development Datasets") from online P2P provider Bondora which contain multiple fields across multiple financial instruments and consumer information was used.

This dataset was subsequently used to build the initial dataset (hereafter referred to as "POA Dataset") which contains the features relevant and corresponds to the narrative set by the business objectives. Thus, the models developed are not predictive of the actual bank consumer base or for any existing unsecured loan application decisioning. The models developed are POCs which serves to highlight the viability and challenges of developing future production models.

1.3 Outline

The report is organised as follows:

- Section 2 introduces the problem definition, some of the methodologies used during the project and the metrics used to evaluate the models.
- Section 3 describes the process of the constructing the POA Dataset from the Development Datasets, the initial features selected for the POA Dataset and the reasons for their selection. The section also describes how the POA Dataset is split into the training and testing datasets.

- Section 4 details the steps of data understanding and data preparation, part of the CRISP-DM strategy used throughout the project. The section describes the findings of the exploratory data analysis on the POA Dataset, and the subsequent preprocessing steps carried out.
- Section 5 describes the various modelling strategies and processes and summarises the various evaluation results obtained on the training dataset.
- Section 6 details the evaluation process and result of the chosen models on the testing dataset. It also discusses about the important features as identified via the various models.
- Section 7 finally discusses about the results, the challenges and issues faced during the projects, possible future works and other considerations.

2 Definitions and Methodology

2.1 Problem Definitions

Given that the goal of the project is to develop a model that can predict the probability of a loan default for thin-file applicants, which can then be used as a basis to develop a risk score, the problem can be treated as a binary probabilistic classification problem, where the target variable would be the occurrence of a default.

Defaulted loans are defined in this project as loans being more than 60 days behind regular payments within 6 months from loan disbursement.

Thin-file applicants are defined as applicants with less than 6 months of credit history/transaction with the bank. Relevant loan applications are thus defined as belonging to loan applicants who have no prior transactions or applications more than 6 months earlier.

2.2 Classification Models

With a binary probabilistic classification problem, four classification models which are able to "estimate" the probability for a loan application to either default or not were chosen. These are:

1. Logistic Regression – A linear model used to predict the log of the odds ratio. It has the advantage of being interpretable, i.e., being able to interpret how each feature contributes positively or negatively to the predicted outcomes. Scikit-learn's `LogisticRegression` implementation was used for the project.

2. Decision Tree – A non-linear/parametric model that is also fairly easy to understand and interpret. It is less affected by multicollinearity, but tends to overfit. Scikit-learn's `DecisionTreeClassifier` implementation was used for the project.

3. Random Forest – An ensemble method based on multiple decision trees, which reduces overfitting, improves stability and accuracy. Also works well with categorical and numerical data. Scikit-learn's `RandomForestClassifier` implementation was used for the project.

4. Gradient Boosted Tree – An ensemble of weak learners where each subsequent weak learner learns from the outcomes of existing weak learners to minimise a loss/error function. A variant of this, XGBoost's `XGBClassifier` was used for the project.

2.3 Cross-validation Strategy

Throughout the modelling process on the train dataset, cross-validation is carried out in order to ensure that validation performances of the models are better representatives of the model's performances when they predict on yet-unseen data (test dataset). Figure 2 depicts the overall modelling and cross-validated evaluation strategy that was adopted throughout the whole project.

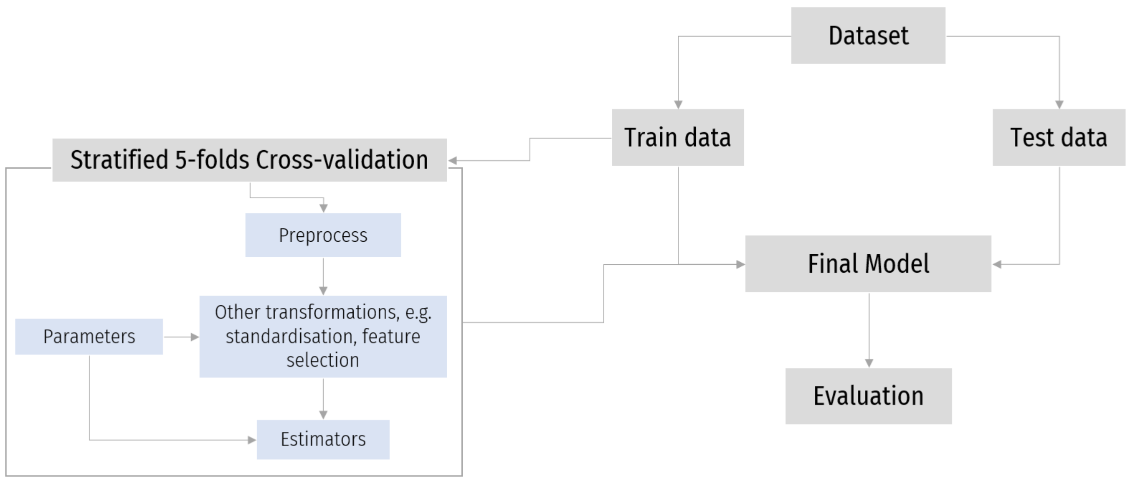

Dataset
Train data
Test data
Stratified 5-folds Cross-validation
Preprocess
Parameters
Other transformations, e.g. standardisation, feature selection
Estimators
Final Model
Evaluation

Figure 2 – Modelling and cross-validated evaluation strategy.

For the project, the `StratifiedKFold` cross-validation iterator from Scikit-learn was used. The stratified strategy allows for each train and validation set within each fold to have approximately the same distribution of samples of each target class (in our case, 1: default, 0: non-default). This helps provide a more accurate performance estimation in our case as we have a slightly imbalanced dataset.

5-folds was chosen arbitrarily, given that it provides a fair balance between processing speed, number of data rows per fold and variance between the different folds. The cross-validation iterator is instantiated as an object and is passed into various Scikit-learn's functions that utilise cross-validation as seen in Figure A-1.

2.4 Hyperparameter Tuning

To further optimise the learning process and thus the performance of the models beyond the default out-of-box settings, hyperparameter tuning was carried out during the project. Three techniques were used throughout the project: `validation_curve`, `RandomizedSearchCV` and `GridSearchCV.`

The first step is to narrow down on the appropriate range of values for a parameter for finer tuning at the next stage, validation curves were plotted using Scikit-learn's `validation_curve` function. A maximum of 4 parameters were tuned for each model, across a range of values that can generated via Numpy's array generator or a custom list of values. An example is provided in Figure A-1.

From the generated validation curve, we are able to narrow down the range of values for the specific parameter that generates the highest scores, which can be used as a reference for the subsequent step. From the derived plot in Figure 3, we can see that the appropriate range of values for the example in Figure A-1 would be somewhere in the range of 50 to 300.

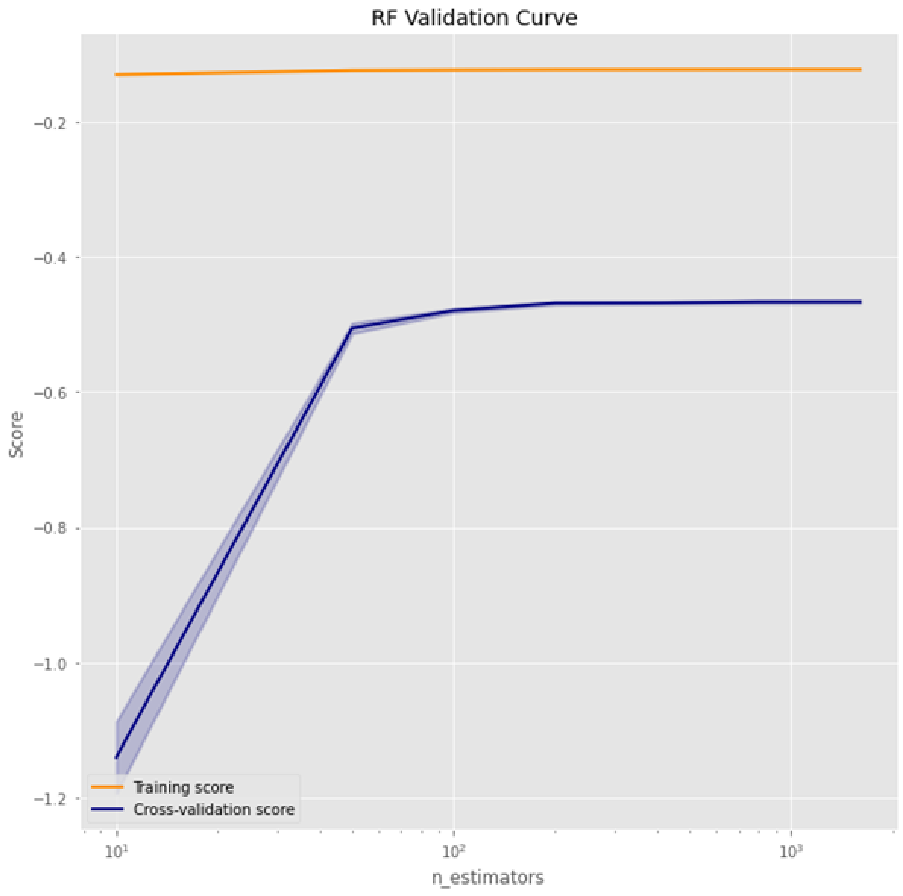
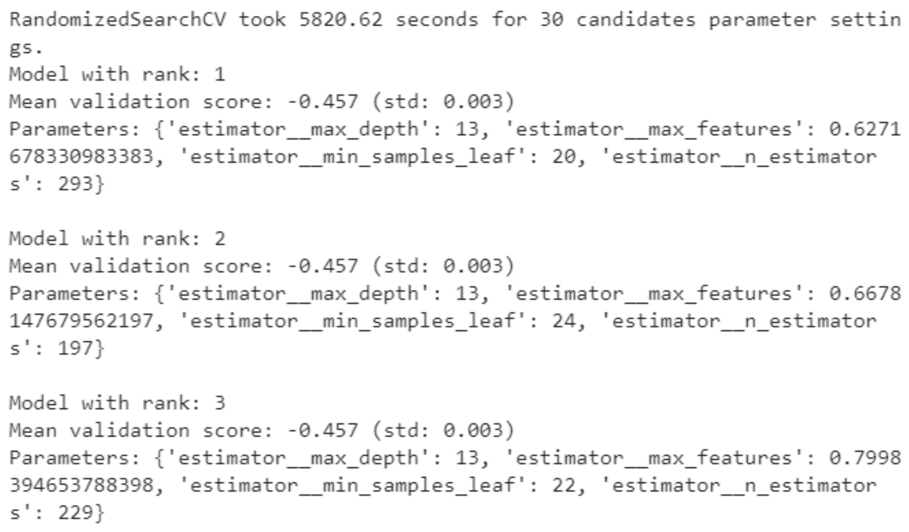


Figure 3 – Validation curve of the n_estimator parameter for the RandomForestClassifier. From the validation score curve, we see that the peak score occurs earliest at around 50 to 300.

Having narrowed the range of values for the various parameters, we carry out a search over the narrowed-down ranges of the various parameters all at once. This was done by either an exhaustive search over the specified parameter ranges via Scikit-learn's `GridSearchCV` or a randomised search over the parameter ranges via `RandomizedSearchCV`. The overall process flow can be seen in Figure 15.

In the example below, randomised search was carried out across four parameters, each of which have been passed through the validation curve technique as above to derive an appropriate range. These range values are then fed as a parameter grid into the `RandomizedSearchCV` as depicted in Figure A-2.

This will return the top few parameter settings from which we can derive the top parameter values as seen in Figure 4.



```
RandomizedSearchCV took 5820.62 seconds for 30 candidates parameter settin
gs.
Model with rank: 1
Mean validation score: -0.457 (std: 0.003)
Parameters: {'estimator__max_depth': 13, 'estimator__max_features': 0.6271
678330983383, 'estimator__min_samples_leaf': 20, 'estimator__n_estimator
s': 293}

Model with rank: 2
Mean validation score: -0.457 (std: 0.003)
Parameters: {'estimator__max_depth': 13, 'estimator__max_features': 0.6678
147679562197, 'estimator__min_samples_leaf': 24, 'estimator__n_estimator
s': 197}

Model with rank: 3
Mean validation score: -0.457 (std: 0.003)
Parameters: {'estimator__max_depth': 13, 'estimator__max_features': 0.7998
394653788398, 'estimator__min_samples_leaf': 22, 'estimator__n_estimator
s': 229}
```

Figure 4 – Top parameter settings returned from the RandomizedSearchCV.

Do note that this method does not necessarily return the best performing model for the current dataset as that would require a substantial amount of minute value optimisation for each parameter. Nevertheless, this method does provide a quality solution with an acceptable runtime.

For GridSearchCV, the process is similar, but becomes more exhaustive and time-consuming with more parameters to tune as the search is exhaustive.

2.4.1 Nested vs Non-nested Cross-validation

The method described above is a non-nested cross-validation strategy, where the same data is used to tune the model parameters and evaluate the performance. Ideally a nested cross-validation (where the cross-validated parameter search is nested within another cross-validated evaluation process) should be adopted to for the hyperparameter search to provide a more generalized score and a better parameter selection [3].

However, the nested cross-validation results in an exponential increase in runtime which the project cannot afford at this stage. As such it is taken in good faith that the parameters selected during the cross-validated parameter search would return approximately the best results. Subsequently, the models with the chosen parameter settings are independently evaluated in a separate cross-validated evaluation process as shown in Figure 2.

2.5 Over and Undersampling

In general, defaults in loans do not occur at the same rate as loans that resolve. This is also the case for the project as seen in Section 3.4. This leads to an imbalance in proportion of the target class label with default labels being the minority and non-default labels being the majority. If the imbalance is severe, the performance of the classification models can be skewed [4].

Though the level of imbalance in our dataset is not severe and is not a concern, resampling methods were explored to understand the effect on the performance of the models developed. For the project, two resampling techniques implemented by the imbalanced-learn package were tried:

1. SMOTE + Random Undersampler – a manual sequential combination of the oversampling SMOTE technique and random undersampling to increase the minority: majority class ratio from about 0.29 to 0.50.

2. SMOTE + ENN – the `SMOTEENN` implementation was used, with default parameters.

Both implementations can be instantiated as a transformer object which can be included into a pipeline for the data to pass through into the final classification estimator. More details on how these are implemented in the project can be found in Section 5.3.

2.6 Probability Calibration

As the project's objective is to develop a binary probabilistic classification model that is capable of predicting the probability of a loan application to default, the probabilistic output of our classifier system should represent reliable probabilities. However, not all classification models/estimators do so and hence probability calibration is necessary [5].

Calibration curves are first plotted using an implementation of the `calibration_curve` function in Scikit-learn to visually compare how well the classifier models are calibrated. Thereafter, the models are calibrated using the `CalibratedClassiferCV` class fitted to either a sigmoid (Platt's) or isotonic regressor and subsequently evaluated.

Figure A-3 shows an example of how the reliability curves can be plotted, Figure A-20 shows an example of the reliability curves. Figure A-4 shows the process of calibrating the various models.

2.7 Evaluation Metrics

This section describes the metrics used to evaluate the models developed throughout the project.

2.7.1 Probability Estimates

Given that the objective of the project is to predict the probability of loan defaults, a number of metrics were used to evaluate the models' performance.

To evaluate the quality of fit of the probability estimates, two metrics – Log Loss, Brier Score were used. The lower these scores are, the better the quality of fit the models have, and thus can be used as a comparative metrics between the models.

Another metric which was used is the ROC AUC which also allows the evaluation of the models' ability to discriminate between positive and negative samples across the entire probability threshold range (maximum to minimum as produced by the classifier).

For calibration and optimisation of models, we will primarily use log loss as the metric to minimise or maximise (in the case of negative log loss in some implementation).

A number of highly informative discussion on the appropriate choice of evaluation metrics can be found on StackExchange [6] [7].

2.7.2 Classification

After the models have been calibrated and optimised through the minimisation of log loss, the appropriate probability threshold is chosen (Section 6.2.1)and subsequently the models' performances for the binary classification task can be evaluated with a number of metrics such as F1-score, precision, recall and Matthew's Correlation Coefficient (MCC).

2.8 Tools and Frameworks

1. **pandas**: pandas is an open-source data analysis and manipulation tool, built on top of the Python programming language [8]. This framework is used throughout the project as it allows for easy manipulation of tabular data.

2. **Scikit-learn**: Scikit-learn or commonly known as Sklearn, is an open-source machine learning library for Python. It has a wide array of implementations of various machine learning techniques [9]. Another widely used framework during the project. Various classification estimators, cross-validation strategies, pipelines constructors, probability calibration and others are used.

3. **Sklearn-pandas**: An open-source module that bridges between Scikit-learn and pandas DataFrames. As Scikit-learn deals predominantly with numpy arrays, manipulation of various features/columns can be difficult without the ability to specifically refer to column names which Sklearn-pandas confer [10]. This framework was largely used during the construction of the preprocessing pipeline.

4. **XGBoost**: An open-source optimised distributed gradient boosting library that implements machine learning algorithms under the gradient boosting framework [11]. Its `XGBClassifier` implementation was used in the project.

5. **Imbalanced-learn:** An open-source library relying on scikit-learn and provides tools when dealing with classification with imbalanced classes. The project uses the `SMOTE`, `RandomUnderSampler`, `SMOTEENN` and `Pipeline` implementation from this library [12].

3 Loan Dataset Construction

Before any modelling can be done, an appropriate dataset that fits to the narrative of the business and data analytics objective earlier defined had to be obtained. Due to privacy concerns, data in production could not be used. Instead, an appropriate dataset (Loan Dataset) had to be constructed from the bank's developmental datasets.

3.1 Data Sources

Out of all the developmental datasets, there are two which are the most relevant: one containing information about loan applications of mixed/unknown origin and the another containing the corresponding repayment transactions.

The former dataset contains about 158,309 rows with 112 variables and includes information on the applicants' personal attributes and details on the loan. The latter dataset is much simpler, containing 2,564,195 rows with only 6 variables and includes repayment transaction information on corresponding loans identified by the loan ID.

3.2 Dataset Rules

To construct the initial POA Dataset, some subsetting rules were set:

1. Applicants should belong to the thin-file, no-file group, i.e., clients who have no or 6 months or less credit records with the financial institute.

   This is carried out by grouping all loan applications by each loan applicants (defined as `username`), finding their earliest loan applications (`loanapplicationstarteddate`) and capturing all corresponding loan applications which are less than 7 months (to avoid rounding issues, e.g., 6 vs 6.5 months) from the earliest application (Figure A-5).

2. The definition of default adopted would be the initial failure to make payments for more than 60 days within 6 months from the loan disbursement. Loans which have defaulted within the timeframe but later changed its status (to either repaid, very late payment or restructured) will still be considered to have defaulted. For loans which have defaulted outside of the timeframe, we would assume that it has not defaulted.

   This is carried out by utilising existing data fields, `defaultdate` and `loandate`. We will define the loan application as being defaulted when the difference between `defaultdate` and `loandate` is less than 7 months (Figure A-6).

3. Ensure that the loans have the necessary runway to reach or exceed the 6 months period so that one can be sure that each loan application has indeed reached its intended status.

   This is carried out by ensuring that the time between the loan disbursement date (`loandate`) and the date of information extraction (13/01/2021) is greater or equal to 7 months (Figure A-7).

3.3 Initial Dataset and Features

Thereafter, we filter out the initial set of features to construct the final POA Dataset. These features were chosen to mimic the data features that can be obtained during the actual loan applications and can be grouped into:

Table 1 – Initial set of features in the constructed POA Dataset.

Feature Groups	**Features**
Target label	defaulted
Loan identifiers	loanid
Personal attributes	username, age, gender, homeownershiptype, languagecode, maritialstatus, country, education, nrofdependants
Employment details	employmentstatus, occupationarea, workexperience, employmentdurationcurrentemployer
Income details	incometotal
Loan details	loanapplicationstarteddate, appliedamount, loanduration,
Credit history details	noofpreviousloansbeforeloan, amountofpreviousloansbeforeloan, priorrepayments, previousearlyrepaymentsbeforeloan, previousearlyrepaymentscountbeforeloan, debttoincome, existingliabilities, liabilitiestotal

Overall, the POA Dataset contains 71,969 records and 28 variables. A quick look at the target label distribution reveals a slightly imbalanced distribution.

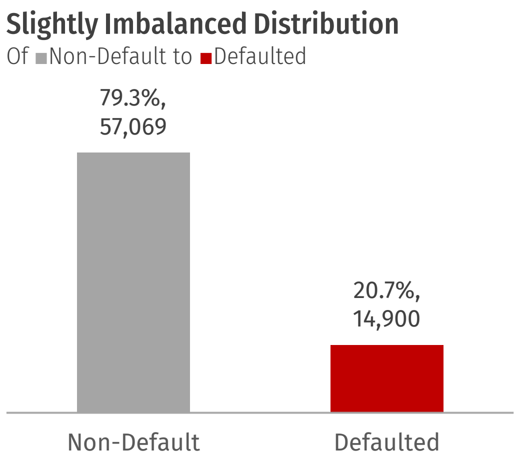


Figure 5 – Imbalanced target label distribution of the POA Dataset.

3.4 Test-Train Split

The next step carried out is to split the POA Dataset into the training and testing dataset. This is carried out at an early stage in order to avoid any potential data leakages. This is carried out by first sorting all the records according to date (again to avoid data leakage) and thereafter splitting the data into an 80:20 ratio (Figure A-8).

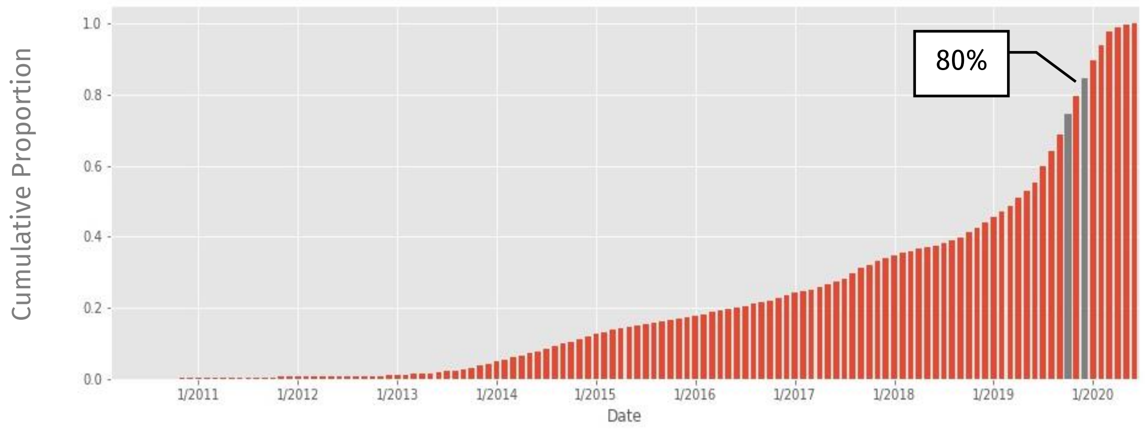


Figure 6 – Train-test split of the POA Dataset into 80:20 (training: testing) ratio.

4 Data Understanding and Preparation

The first step before modelling in accordance to CRISP-DM is to understand the data and then prepare the data. In this section, we describe the exploratory data analysis (EDA) and preprocessing carried out on the training data.

4.1 Exploratory Data Analysis

EDA was carried out on the training dataset which contained 57575 records.

4.1.1 Dependent, Independent Features

The dependent feature, `defaulted` is a categorical variable, with values 1 (defaulted) or 0 (non-default).

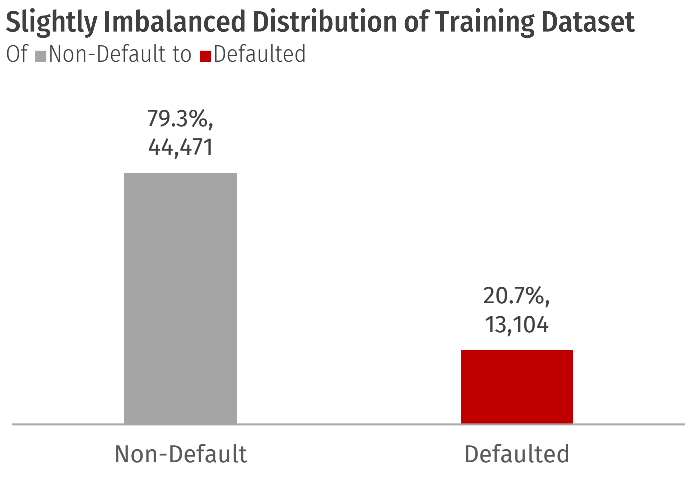


Figure 7 – Imbalanced target label distribution of the training dataset

The independent variables can be categorised into numerical, categorical ordinal and categorical nominal as follows:

Table 2 – Different types of variables in the training dataset.

Feature Groups	Features
Numerical	age, appliedamount, loanduration, incometotal, liabilitiestotal, existingliabilities, debttoincome, noofpreviousloansbeforeloan, amountofpreviousloansbeforeloan, priorrepayments, previousearlyrepaymentsbeforeloan, previousearlyrepaymentscountbeforeloan
Categorical Ordinal	nrofdependants, education, employmentdurationcurrentemployer, workexperience
Categorical Nominal	gender, homeownershiptype, languagecode, maritialstatus, country, employmentstatus, occupationarea

4.1.2 Missing and Undefined Values

One of the first thing explored in EDA is missingness and the presence of undefined values across the variables. Missingness is rather easy to quantify with many Python packages such as missingno [13] which produces missing values visualisations, or with the native pandas methods `isna` (Figure A-9).

For undefined values, we refer to the data dictionary provided by the bank to understand the range of acceptable/defined values for each variable. This occurs with categorical variables. Figure A-10 shows how the undefined values can be summarised for each variable.

We found that there were six variables which have undefined or missing values exceeding 50%:

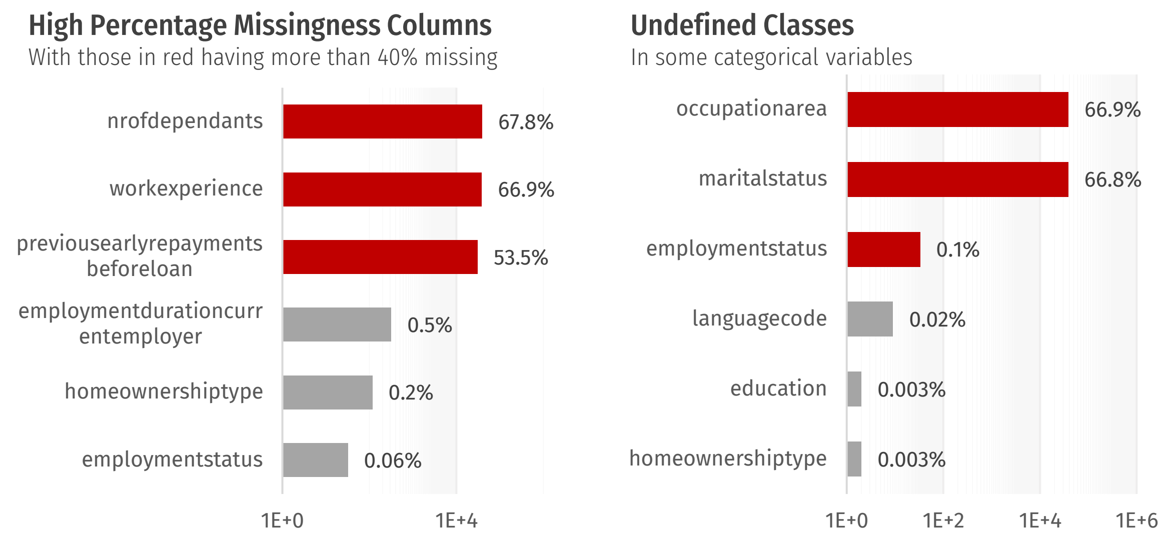


Figure 8 – Percentage of missing values (left) and undefined values (right)

For the treatment of missing values, we first attempted to understand if the missing values are missing at random (MAR), missing completely at random (MCAR) or missing not at random (MNAR).

We quickly found this to be a difficult task especially if one lacks the subject matter expertise and understanding of the data collection process. We attempted to perform statistical tests (Figure A-11: Chi-square and Cramer's V) to investigate if the presence/absence of the missing data in each variable is dependent on the target variable in order to understand the nature of the missingness. However, the tests are inconclusive.

As such, in variables with huge proportion of missing values, e.g., greater than 50%, we assume that the variables are MAR and we thus impute these missing values with a separate class label. For those variables with very small number of missing values, i.e., less than 1%, we assume that they are MCAR and various other imputation strategies e.g., mode, mean, constant etc. are used (more details in Section 4.2.3).

For undefined values, the treatment is rather similar to that of missingness. We impute with a separate class those with a large proportion of undefined values. Those with small occurrences of undefined values were treated similarly as their missing value counterparts. In essence, for undefined values, we effectively assume that they are equivalent to being missing.

4.1.3 Correlation

Next, a quick correlation test visualisation was carried out to understand if there are any strong predictors of the target variable or if there are any potential multicollinearity issues between the independent variables.

For numerical variables, a Pearson's correlation heatmap (Figure 9) was plotted, whilst a Cramer's V heatmap (Figure A-12) was plotted for categorical variables.

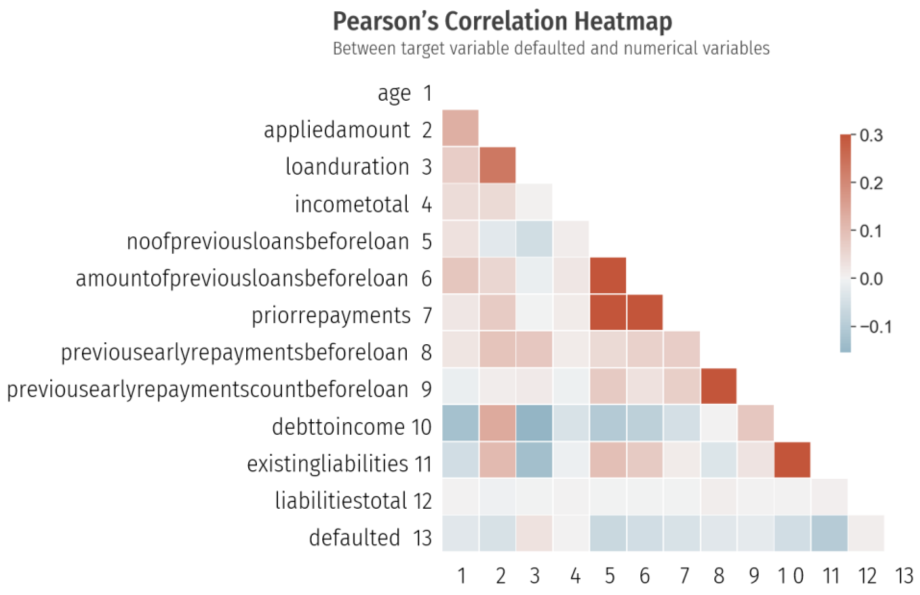


Figure 9 – Pearson's correlation heatmap. Independent variables show weak correlation to the target variable defaulted. Multicollinearity may not be an issue as the correlation coefficient is relatively small at < 0.3.

`loanduration`, and `languagecode` and `country` seems to have the highest positive correlation to the target label out of all the numerical and categorical variables respectively. However, overall, we do not observe strong trends or correlations between the independent variables and the dependent variables. We do see instances of possible multicollinearity, which we will leave untreated as they seem weak.

4.1.4 Univariate Analysis

Subsequently, univariate analysis on each variable was carried out. We explored the distributions of each variable independently and by the target labels. We also investigated their correlation strength with the target variable and explored methods to transform the variables to improve this correlation or reduce distribution skewness (if any).

For each numerical variable, we plotted a frequency histogram (for continuous numerical) or bar chart (for discrete numerical), a Q-Q plot, boxplots grouped by the target label and a kernel density estimate (KDE) plot also grouped by the target label. We calculated the skewness and also the Pearson's correlation of the variable to the target variable. See Figure A-13 for details on the code and Figure 10 for an example of the plots.

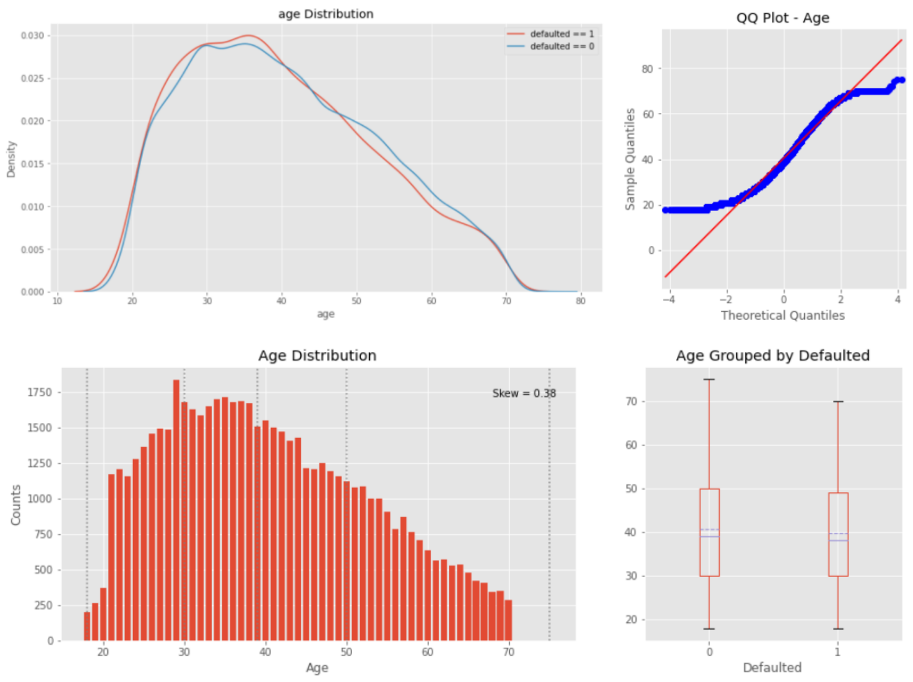


Figure 10 – Diagnostic plots for the discrete numerical variable age showing a slight skewness of 0.38

For categorical variables we plot bar charts for the variables, grouped by the target label. We also calculated the Cramer's V and Pearson's correlation coefficient:

```
Cramer's V = 0.04375325737270125
Pearson Chi-squared
probability = 0.950, critical = 11.070, stat = 42.699
Dependent (reject H0)
significance = 0.050, p = 0.000
Dependent (reject H0)
```

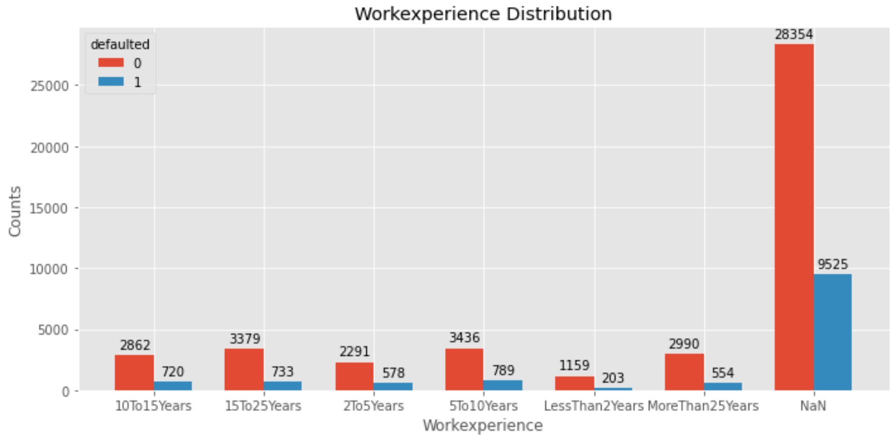


Figure 11 – Bar plot for the categorical numerical variable workexperience showing weak association with the target variable

Overall, we did not find any interesting trends not already observed from the earlier correlation heatmap. Nevertheless, these plots are useful for understanding the distributions of the variables, how the missing or undefined values are distributed and also to visualise how subsequent transformations affect the distribution and the correlation to the target label.

We do observe that there are a number of variables (`noofpreviousloansbeforeloan`, `amountofpreviousloansbeforeloan` and `priorrepayments`) where there is a huge proportion of zeros, which may potentially lead to a zero-inflation problem.

4.1.5 Outliers

We considered removing records which are potentially outliers. However, as there was a lack of subject matter experience and the actual context of the original dataset, it is difficult to differentiate if the values are outliers/anomalies are just extreme values. As such we will keep all the records as they were.

Given that we have structured all our processing steps into a pipeline that will be passed as part of our model we could pass an intermediate outlier detection and removal step if and when necessary without much interruption.

4.2 Data Preprocessing

4.2.1 Preprocessing Pipeline

For the project, the data preprocessing steps would be integrated into as part of the entire modelling pipeline. The modular structure of the modelling pipeline allows easy manipulation and more importantly, avoids data leakage and ensures consistency and reproducibility.

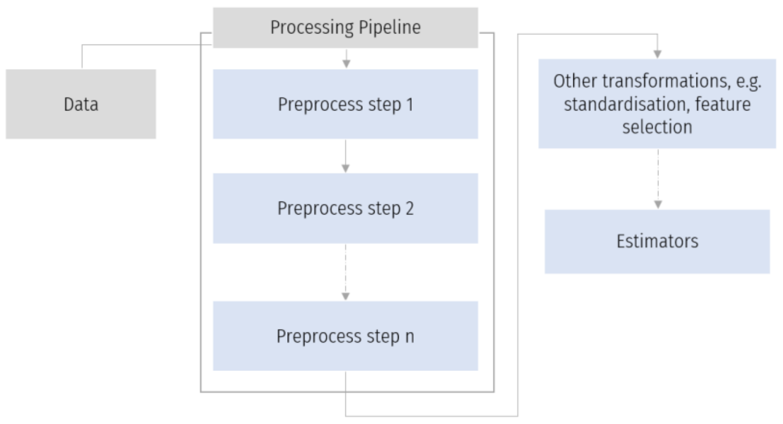


Figure 12 – Processing pipeline schematic

In the following section, some of the steps included within the preprocessing pipline are described. Refer to Figure A-15 to Figure A-17for a snapshot of the entire pipeline code.

4.2.2 Type Casting

One of the most important steps in preprocessing is the data type casting. Type casting resolves 2 major issues: 1) Scikit-learn's estimator functions can only take in numerical values in variables to be meaningful; 2) Certain variables contains undefined values (as described earlier).

In the project, numerical variables are first type casted as float type, while categorial variables are type casted as using the `CategoricalDtype` instance which allow the manipulation of the classes/categories acceptable in a variable field and if the variable is order or unordered. Those class/category values which are not defined, will then be revalued as NaN which can then be treated accordingly. This corresponds to the strategy earlier stated in Section 4.1.2.

4.2.3 Missing Data Imputation

To resolve the existing NaN values, a variety of imputation methods were carried out. For categorical variables which have missing values more than 50%, they were imputed with the constant value -1 or "-1", while others with low occurrences are imputed with the mode/most-frequent class values.

For the numerical variable `previousearlyrepaymentsbeforeloan` which has more than 50% undefined values, we found out that a majority of them can be imputed as 0 as it is necessarily so when the `previousearlyrepaymentscountbeforeloan` is 0. This was carried out by a custom impute function. We then grouped all the records according to the `previousearlyrepaymentscountbeforeloan` and found the corresponding median value of `previousearlyrepaymentsbeforeloan` for each group. We then imputed the remaining missing values with these median values according to their respective `previousearlyrepaymentscountbeforeloan`. This was also carried out by a custom function.

4.2.4 Feature Engineering

One of the last few steps in the preprocessing pipeline is the feature engineering processes. These include:

- Transformation of the numerical variables – log1p, yeo-johnson, binning and quantile transformations were carried out to reduce the skewness and improve the correlation of the variables to the target variable. An example of the improvement of the `age` variable in skewness and correlation can be seen in Figure 13. This was carried out by various transformer functions in Scikit-learn.

- Encoding of categorical data – Ordinal encoding of categorical ordinal variables using Scikit-learn's `OrdinalEncoder` function with the order earlier defined by the data type casting process, and One-hot encoding (drop first column) of categorical nominal variables using Scikit-learn's `OneHotEncoder`.

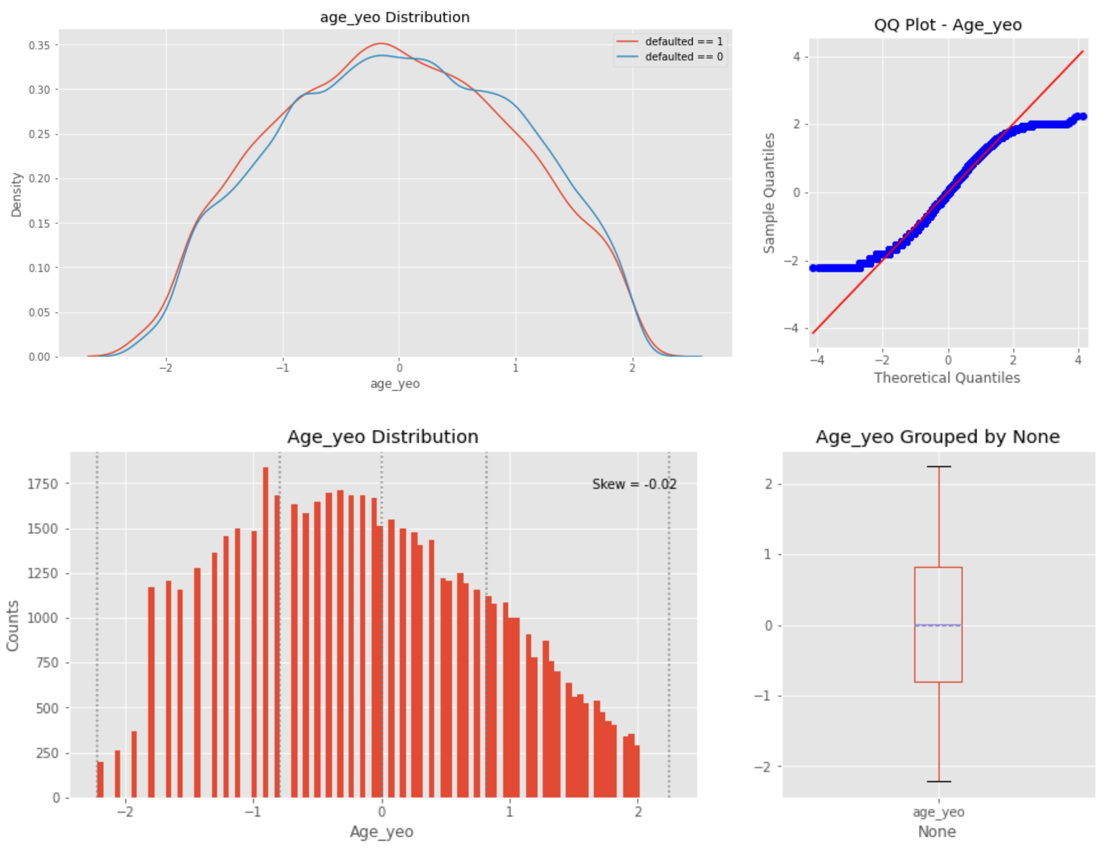


Figure 13 – Diagnostic plots of the age variable after Yeo-Johnson transformation. Skewness is reduced from 0.38 to 0.02 and correlation from 0.0256 to 0.0257.

5 Modelling

Having constructed the preprocessing pipeline, the main modelling work can be carried out. Here we use the cross-validation strategy depicted in Figure 2.

Various methods were experimented to improve the performance of the classification model and the outcomes are evaluated on the validation dataset, i.e., the results reported in this section are those derived from the cross-validation evaluation on the training dataset.

5.1 Baseline Models & Default Classification Models

To evaluate the models created, we first established baseline models whose performance would serve as the minimally achievable performance that our classification models should exceed.

3 baseline models were established via Scikit-learn's DummyClassifier, with the following classification strategy:

- Stratified - Generates random predictions, while respecting class distribution
- Most frequent – Always predicts the most frequent class for all records
- Prior – Predicts the most frequent class and returns the class prior probabilities

At the same time, the Logistic Regression, Decision Tree Classifier, Random Forest Classifier and XGBClassifiers were also instantiated on default settings.

5.1.1 Setup

The various models were set up as follows:

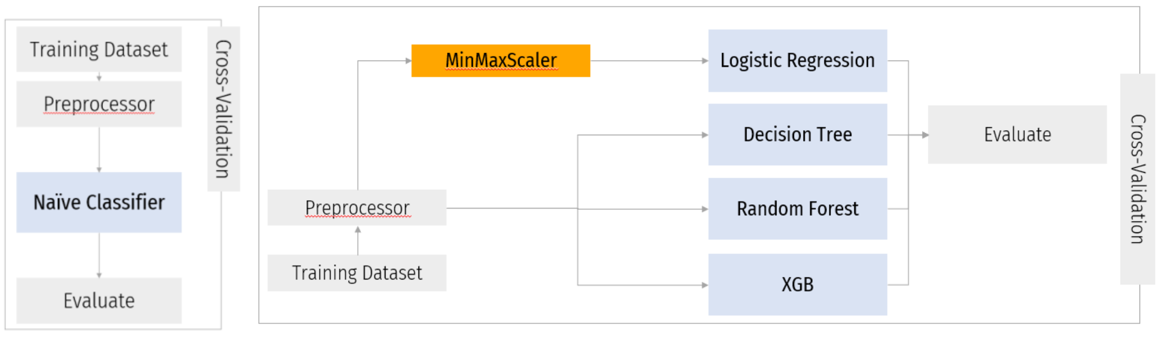


Figure 14 – Model set-up for the baseline (naïve) and default classifiers.

For the logistic regression model, standardisation was carried out via the use of a `MinMaxScaler` This brings all the variables onto the same scale so that feature importance based on the coefficients can be effectively compared across all variables. `MinMaxScaler` was used instead of `StandardScaler` in order to preserve the shape of the dataset. This could potentially allow for outlier removal methods to be used in the future.

5.1.2 Cross-Validation Results

The cross-validation results on the training dataset are as follows:

Table 3 – Cross-validation result of the baseline and default models.

Model	**ROC AUC**	**Neg. Log Loss**	**Neg. Brier Score**
Naïve – Stratified	0.498	-12.084	-0.350
Naïve – Most frequent	0.500	-7.861	-0.228
Naïve – Prior	0.500	-0.536	-0.176
LR – Default	0.735	-0.473	-0.154
DT – Default	0.582	-10.427	-0.302
RF – Default	0.730	-0.486	-0.155
XGB - Default	**0.745**	**-0.469**	**-0.151**

Learning curves of the various default models were also plotted in order to understand how the models can be better tuned (Figure A-18).

5.2 Hyperparameter Tuning

Hyperparameter tuning was then carried out to further optimise the models. In order to figure out which parameters to optimise, the learning curves were studied.

From the logistic regression learning curves, we see that the lines converge, indicating that the model had been fitted rather well. However, the overall high bias suggested that the model may not be complex enough fit the data.

For the decision tree, we see that the training curves remain constant at a rather high score while the cross-validation curve remains low and rises very slowly. The 2 curves do not converge. Overall, it was interpreted that the model had high bias and high variance and it is likely because of overfitting. Parameters (such as max_depth, max_features, min_impurity_decrease) can be optimised to control the complexity of the model to generalise and to improve the performance.

For the default random forest, we see an improvement compared to the decision tree, however, a based on the curves, we concluded that overfitting occurs too. Parameters such as n_estimators can be optimised to reduce the complexity and increase model performance.

For the default XGB model, we observe that the training and validation learning curves are trending towards convergence. The model has slightly high bias and relatively low variance. Given that XGB is highly optimisable, parameters controlling the complexity were optimised, in hopes to increase complexity but control overfitting at the same time.

5.2.1 Setup

The set of parameters for each model that were optimised are as described in Figure 15, with the corresponding range of values that were searched across. Figure 15 also shows the process flow by which the optimisation was carried out.

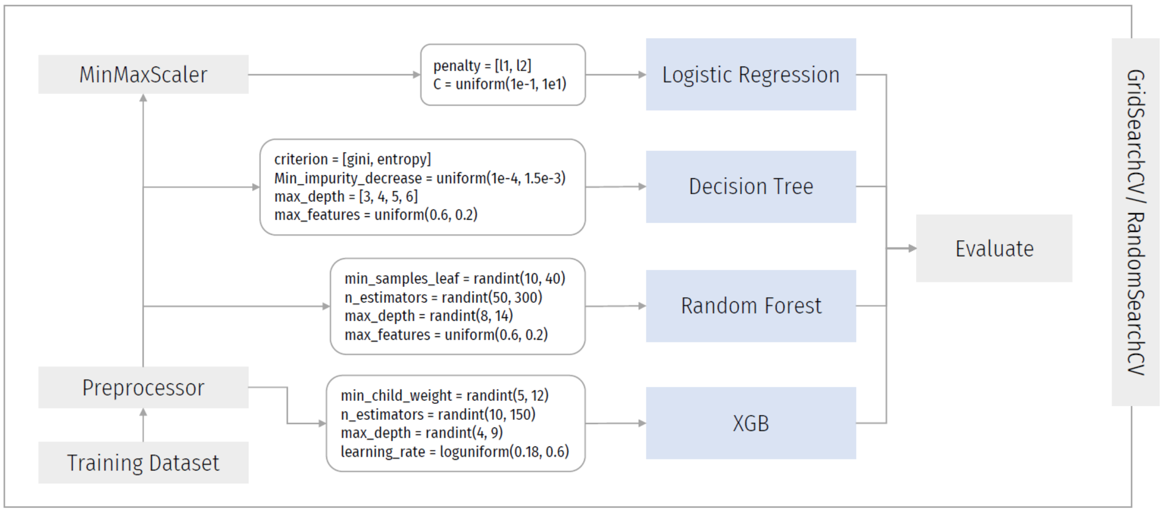

MinMaxScaler
penalty = [l1, l2]
C = uniform(1e-1, 1e1)
Logistic Regression
criterion = [gini, entropy]
Min_impurity_decrease = uniform(1e-4, 1.5e-3)
max_depth = [3, 4, 5, 6]
max_features = uniform(0.6, 0.2)
Decision Tree
Evaluate
min_samples_leaf = randint(10, 40)
n_estimators = randint(50, 300)
max_depth = randint(8, 14)
max_features = uniform(0.6, 0.2)
Random Forest
Preprocessor
min_child_weight = randint(5, 12)
n_estimators = randint(10, 150)
max_depth = randint(4, 9)
learning_rate = loguniform(0.18, 0.6)
XGB
Training Dataset
GridSearchCV / RandomSearchCV

Figure 15 – Hyperparameter tuning via GridSearchCV or RandomSearchCV. Cross-validated search over a range of values for some parameters as carried out for the various classification models.

5.2.2 Cross-Validation Results

Table 4 shows the cross-validated results of the tuned model against the default models

Table 4 – Cross-validation result of the tuned models.

Model	**ROC AUC**	**Neg. Log Loss**	**Neg. Brier Score**
LR – Default	0.735	-0.473	-0.154
LR – Tuned	0.735	-0.473	-0.154
DT – Default	0.582	-10.427	-0.302
DT – Tuned	0.725	-0.477	-0.155
RF – Default	0.730	-0.486	-0.155
RF – Tuned	0.752	-0.462	-0.150
XGB - Default	0.745	-0.469	-0.151
XGB - Tuned	**0.754**	**-0.461**	**-0.149**

Figure A-19 shows the various learning curves after tuning.

5.3 Class Imbalance Treatment

As there was slight imbalance in the target distribution in the training dataset, it was hoped that by training the models on a more balanced dataset could help improve the performance. To that end, 2 methods of combined over and undersampling were carried out (as described in Section 2.5).

5.3.1 Setup

Due to issues with pipeline compatibility (there were some issues syncing the pipeline from Scikit-learn used in the preprocessing steps with the pipeline from imbalanced-learn), we were unable to integrate the under and oversampler steps together with the preprocessing steps in a single pipeline.

As such, we attempted to investigate if sampling improves performance by first preprocessing the training dataset outside the cross-validation cycle and passing the output dataset into the cross-validated sampling and estimator (on default settings) pipeline. To provide a fair comparison, the same procedure was carried out on the cross-validated estimator (on default settings) pipeline without sampling.

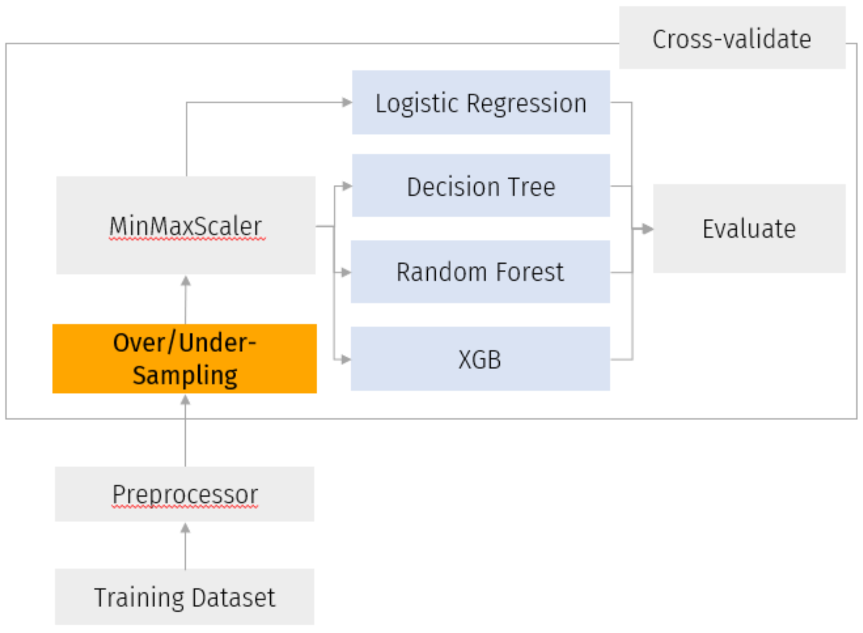


Figure 16 – Pipeline of how over and undersampling techniques were tried and evaluated

5.3.2 Cross-Validation Results

Table 5 shows the results of the above experiment.

Table 5 – Cross-validation result of the SMOTEENN models, the SMOTE RUS models (SMOTE and random undersampler) and the vanilla models (default estimator with no sampling).

Model	ROC AUC	Neg. Log Loss	Neg. Brier Score
LR – Default vanilla	0.735	-0.473	-0.154
LR– Default SMOTE RUS	0.735	-0.496	-0.163
LR– Default SMOTEENN	0.732	-0.856	-0.291
DT– Default vanilla	0.582	-10.399	-0.301
DT– Default SMOTE RUS	0.592	-10.970	-0.318
DT– Default SMOTEENN	0.631	-11.884	-0.344
RF– Default vanilla	0.730	-0.487	-0.155
RF– Default SMOTE RUS	0.733	-0.491	-0.159
RF– Default SMOTEENN	0.739	-0.761	-0.213
XGB– Default vanilla	0.746	-0.469	-0.151
XGB– Default SMOTE RUS	0.746	-0.474	-0.154
XGB– Default SMOTEENN	0.744	-0.614	-0.202

5.4 Probability Calibration

Probability calibration was carried out on the four tuned classification models. The intent is to allow for the model to provide a better probability estimation.

Reliability plots of the probabilistic predictions before and after calibration (Figure A-19 and Figure A-20) were made to compare how well the calibrations were, and the calibrated models were also evaluated on the training dataset to provide a comparison.

5.4.1 Setup

Scikit-learns `CalibratedClassifierCV` was used for the calibration. It acts like a meta-estimator and is used by wrapping it around the base estimator within the modelling pipeline.

5.4.2 Cross-Validation Results

Table 6 – Cross-validation result of the calibrated models. Two calibration methods were used, Sigmoid (Platt's) and Isotonic.

Model	**ROC AUC**	**Neg. Log Loss**	**Neg. Brier Score**
LR – Tuned	0.735	-0.473	-0.154
LR – Tuned Sigmoid	0.735	-0.473	-0.154
LR – Tuned Isotonic	0.735	-0.473	-0.154
DT – Tuned	0.725	-0.477	-0.155
DT – Tuned Sigmoid	0.734	-0.476	-0.154
DT – Tuned Isotonic	0.734	-0.474	-0.153
RF – Tuned	0.752	-0.462	-0.150
RF – Tuned Sigmoid	0.752	-0.465	-0.150
RF – Tuned Isotonic	0.752	-0.462	-0.150
XGB – Tuned	0.754	-0.461	-0.149
XGB – Tuned Sigmoid	0.755	-0.464	-0.150
XGB – Tuned Isotonic	**0.755**	**-0.460**	**-0.149**

6 Final Evaluation

We consolidated the top performing models across the different model class types and strategies in the list below:

- XGB – Tuned Isotonic
- RF – Tuned
- LR – Tuned
- DT – Tuned
- Naïve Classifiers

6.1 Probability Estimation Evaluation

We re-instantiated the models, trained them on the training dataset and carried out evaluation on the testing dataset which has been left untouched since splitting. Table 7 shows the results of the evaluation:

Table 7 – Evaluation results of the LR-tuned, DT-tuned, RF-tuned, XGB-tuned-isotonic and naïve classifiers on training dataset.

Model	**ROC AUC**	**Log Loss**	**Brier Score**
Naïve – Stratified	10.157193	0.294081	0.498761
Naïve – Most frequent	0.500000	4.309549	0.124774
Naïve – Prior	0.500000	0.410715	0.119779
LR – Tuned	0.724631	0.371227	0.110702
DT – Tuned	0.709649	0.366089	0.107761
RF – Tuned	0.721236	0.353999	0.104156
XGB – Tuned Isotonic	**0.734359**	**0.346553**	**0.102394**

Note that with log loss and Brier score, a smaller value is better. This is opposite of their negative counterparts earlier used, where the larger they are, the better.

6.2 Classification Performance

To evaluate the classification performance, we will make use of the probability estimation provided by the classifier models which represents the probabilities a particular loan application should have a value of 1 (default) and 0 (non-default). In essence, it signifies the risk of default for a particular loan application. To classify the outputs into 1/0, a particular threshold would be required.

6.2.1 Threshold Optimisation

In a normal setting, the company would have a decision function which takes in these probabilities and outputs this threshold and thus the decision on whether to classify the loans as default or non-default based on the relative risk appetite and costs. As these costs and risks are dependent on the company, the product and market segment themselves, for this project, we decided to use a standardised/neutral method to optimise this threshold.

We attempted to find the best probability threshold for each model that maximises the Geometric Mean (G-Mean) of the sensitivity and specificity, which is better than maximising for accuracy for an imbalanced dataset [14]:

$$G - Mean = \sqrt{Sensitivity \times Specificity} = \sqrt{TPR \times (1 - FPR)}$$

Figure A-22 and Figure A-23 shows the individual ROC AUC curves of the four classifier models and the relative probability threshold point and value which maximises the G-Means.

6.3 Classification Evaluation

With the optimised threshold, we carried out an evaluation and carried out model evaluation using the metrics defined earlier in Section 2.7.2.:

Table 8 – Evaluation results of the LR-tuned, DT-tuned, RF-tuned, XGB-tuned-isotonic and naïve classifiers on training dataset. Naïve prior and Naïve most frequent models are left out as they always predict the most frequent class label, result in a zero TP and FP which leads to ill-defined or 0 value metrics

Model	MCC	F1	Precision	Recall
Naïve – Stratified	-0.001961	0.158951	0.123571	0.222717
LR – Tuned	0.234272	0.335621	0.221295	**0.694321**
DT – Tuned	0.190831	0.311643	0.238546	0.449332
RF – Tuned	0.235409	0.345528	**0.278554**	0.454900
XGB – Tuned Isotonic	**0.241395**	**0.351035**	0.270295	0.500557

Here we see that XGB – Tuned Isotonic has the best MCC and F1 scores, which are better indicators of classification performance than precision and recall in an imbalanced dataset.

Overall, it does seem that the XGB – Tuned Isotonic model outperforms the others.

6.4 Feature Importance

From the XGB -Tuned Isotonic model, we obtained the feature importance by gain Figure A-25 or weight Figure A-26. The models consider the One-hot encoded features and transformed features to be separate features from the underlying base feature of the original dataset. Whilst statistically true, in order to improve interpretability for the business context, we renamed each feature back to their underlying base features and took the top scoring instance for renamed feature to rank them. For example: country_x0_ES and country_x0_FI would both be renamed as the base feature country and we will retain the renamed country_x0_ES as it has a higher score on the gain-based feature importance chart. The top features according to XGB-Tuned Isotonic are as follows:

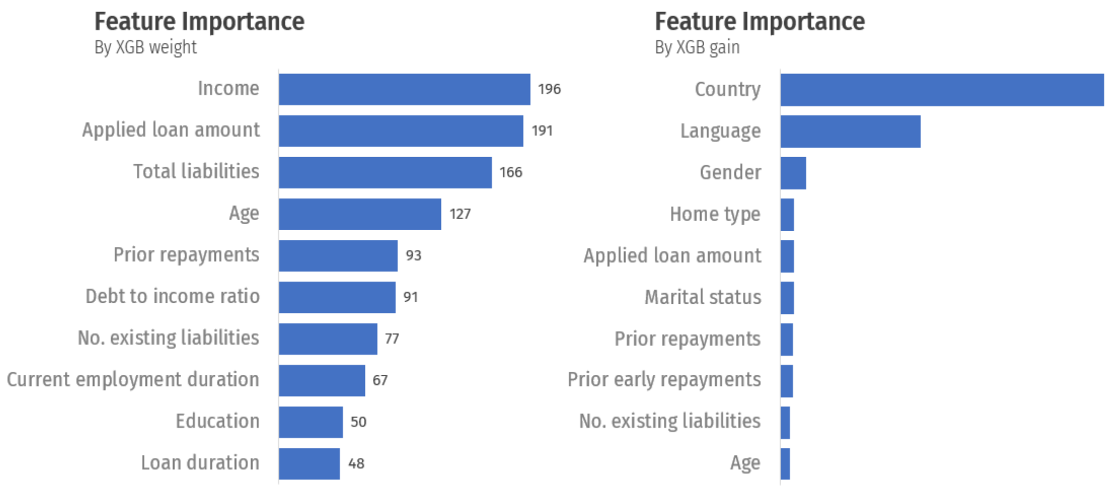


Figure 17 – Top feature importances according the XGB – Tuned Isotonic model by weight (left) and gain (right).

The top features according to the LR-Tuned, DT-Tuned, RF-Tuned models are also consolidated in Figure A-27 and Figure A-28.

7 Discussion and Conclusion

7.1 Discussion

7.1.1 Model Performance

It was observed from Table 3 that out of the box, the Logistic Regression (LR), Random Forest (RF) and XGBoostClassifier (XGB) models worked much better than the Naïve classifiers, with XGB-Default having the best performance (49%, 12.5%, 14.2% better than naïve models in ROC AUC, log loss, Brier score respectively). The Decision Tree classifier (DT) does have a much smaller neg. log loss compared to the naïve models and is due to overfitting as concluded from the learning curves (Figure A-18).

To further improve the performance, parameter tuning was carried out. From Table 4 it was observed that XGB-Tuned stays ahead of the pack, having improved performance over the naïve models (ROC AUC: 50.8%, log loss: 14.0, Brier score: 15.3%). We see a huge improvement in the DT model and moderate improvements in the RF and XGB models after tuning, signifying that the tuning strategy seems to at least be in the correct direction. As expected, there is limited improvement in the LR model due to the lack of model complexity capability.

Experiments with sampling was carried out to understand if such strategies can help improve predictions. The experiment setup is described in Section 5.3.1. When compared to the similarly processed vanilla models (Table 5), the probabilistic estimation performance of the models does not seem to improve, but diminishes instead as evident from the poorer neg. log loss and Brier score metrics of the SMOTEENN and SMOTE-RUS models. ROC AUC seems to improve for the RF and DT models with under and oversampling and this is because these default models tend to overfit the training data and may have learnt to over predict a particular target class label which could theoretically improve ROC AUC, but in reality, provides a poor probability estimate as indicated by the porter log loss and Brier score.

In general, we found that the SMOTE-RUS method seems to perform better than the SMOTEENN method, resulting in an average of 0.5%, 3% and 4% reduction in ROC AUC, log loss, Brier score respectively as compared to 2.2%, 45.6% 43.6% reduction for SMOTEENN. The difference could likely stem from the difference in the majority: minority ratio after sampling as SMOTEENN was carried out on default settings. Nevertheless, sampling did not improve our model performances.

As such, with the tuned models, we carried out probability calibration to further improve the probability estimates in Section 5.4. Unlike with sampling, calibration produces a mix bag of results (Table 6). We find limited/no observable improvements in performance for the LR and RF models. For DT model, we find that the calibration improved the performances across three metrics with isotonic calibration being the best. For XGB, the isotonic calibration produces a slight improvement over the underlying tuned XGB model and is overall the best model. It performed 51%, 14.1% and 15.3% better than the best naïve classifier in terms of ROC AUC, log loss and Brier score based on cross-validated evaluation results on the training dataset

Based on testing dataset evaluation, we can conclude that the XGB – Tuned Isotonic model performs the best. It achieved 46.9%, 15.6% and 14.5% increase in probability estimation performance over naïve classifiers based on the ROC AUC, log loss and Brier score metrics. In terms of classification performance, it performed 120% better than the naïve classifier based on F1 scores, when threshold optimised to maximise the G-Means of sensitivity and specificity.

7.2 Challenges and Future Work

7.2.1 Feature Importance

While the classification and probabilistic estimation performances of the developed models are optimistic, there are significant challenges in interpreting the feature importance. Tree-based models such as the DT, RF and XGB all indicated that `country` is of top importance. It is not clear why and if there are confounding factors such as different economic situation, access to alternative credit sources and cultural views on credit in the different countries, which could lead to a potentially spurious association between `country` and default rates. Further investigation should be carried out.

7.2.2 Other Challenges

We also observe that there may be possible multicollinearity issues in the models between the transformed features. The presence of multicollinearity could affect the overall stability of the models. To improve on this, a feature selection step prior to the estimator could be developed to remove collinear features. This requires a number of substeps: 1) identifying and grouping the collinear features, 2) removing some or all but one feature from each group of collinear features based on the significance of the retained features on the prediction performance, 3) integrating this into the modelling pipeline.

7.2.3 Future Work

Another area to work on in the future or if I could redo the project is to carry out more feature engineering prior to modelling. For example, carry out unsupervised clustering of loan applications based on their profile features or factor analysis and use this as another feature input to the model.

Another area to consider would be to perform multivariate analysis during the EDA process as this may highlight certain trends that could help improve the modelling process.

Lastly, it would be to have a deeper engagement with the relevant stakeholders to better understand the business context to facilitate feature engineering and interpretation of the feature importance.

7.3 Conclusion

Overall, through the project, we’ve managed to develop a gradient boosted tree based binary probabilistic classification model, as implemented by XGBoost, that outperforms baseline naïve classifiers and three other candidate models: LogisticRegression, DecisionTreeClassifier and RandomForestClassifier implemented by Scikit-learn. The final model was an XGBClassifier model that was tuned, and calibrated via isotonic regression. In terms of probabilistic prediction, it performed 46.9%, 15.6% and 14.5% better on the testing dataset evaluation than naïve classifiers based on the ROC AUC, log loss and Brier score metrics. In terms of classification performance, it performed 120% better than the naïve classifier based on F1 scores when G-Means (sensitivity/specificity) optimised.

References

[1] Veritas Consortium, “Veritas Document 2 - FEAT Fairness Principles Assessment Case Studies,” Singapore, 2020.

[2] Financial Times, “Singapore’s new digital banks face uphill struggle,” 22 January 2020. [Online]. Available: https://www.ft.com/content/d0560892-36a8-11ea-a6d3-9a26f8c3cba4.

[3] Scikit-learn, “Nested versus non-nested cross-validation,” [Online]. Available: https://scikit-learn.org/stable/auto_examples/model_selection/plot_nested_cross_validation_iris.html?highlight=nested. [Accessed 17 March 2020].

[4] J. L. Leevy, T. M. Khoshgoftaar, R. A. Bauder and N. Seliya, “A survey on addressing high-class imbalance in big data,” *Journal of Big Data,* no. 5, 2018.

[5] C. F. Dormann, “Calibration of probability predictions from machine-learning and statistical models,” *Global Ecology and Biogeography,* vol. 29, no. 4, pp. 760-765, 2020.

[6] S. Kolassa, “Why is accuracy not the best measure for assessing classification models?,” [Online]. Available: https://stats.stackexchange.com/q/312787.

[7] stat2739, “Brier Score and extreme class imbalance,” [Online]. Available: https://stats.stackexchange.com/q/489106.

[8] The pandas development team, “pandas-dev/pandas: Pandas,” Zenodo, 2020.

[9] L. Buitinck , G. Louppe , M. Blondel , F. Pedregosa , A. Mueller , O. Grisel , V. Niculae , P. Prettenhofer , A. Alexandre , J. Grobler , R. Layton , J. VanderPlas , A. Joly , B. Holt and G. Varoquaux, “API design for machine learning software: experiences from the scikit-learn,” 2013.

[10] e. a. Israel Saeta Pérez, “Sklearn-pandas,” [Online]. Available: https://github.com/scikit-learn-contrib/sklearn-pandas#readme.

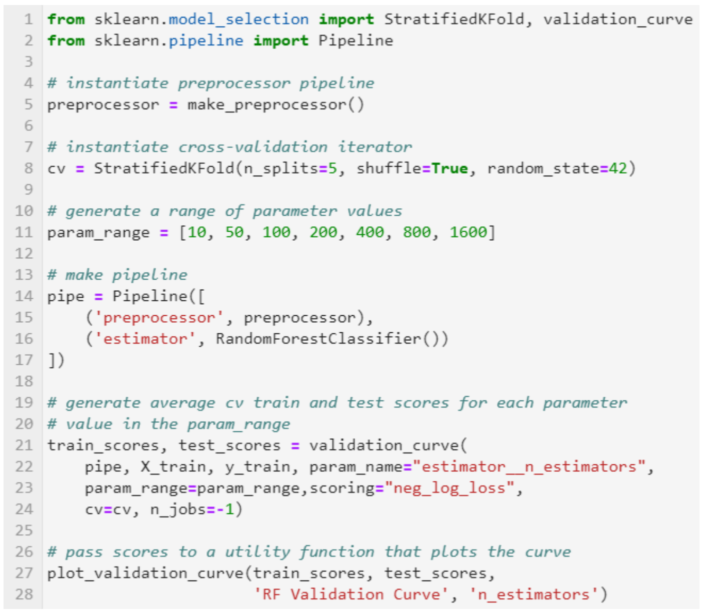
[11] T. Chen and C. Guestrin, “XGBoost: A Scalable Tree Boosting System,” in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining,* San Francisco, California, USA, 2016.

[12] G. Lemaitre, F. Nogueira and C. K. Aridas, “Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning,” *Journal of Machine Learning Research,* vol. 18, no. 17, pp. 1-5, 2017.

[13] A. Bilogur, “Missingno: a missing data visualization suite,” *Journal of Open Source Software,* vol. 3, no. 22, 2018.

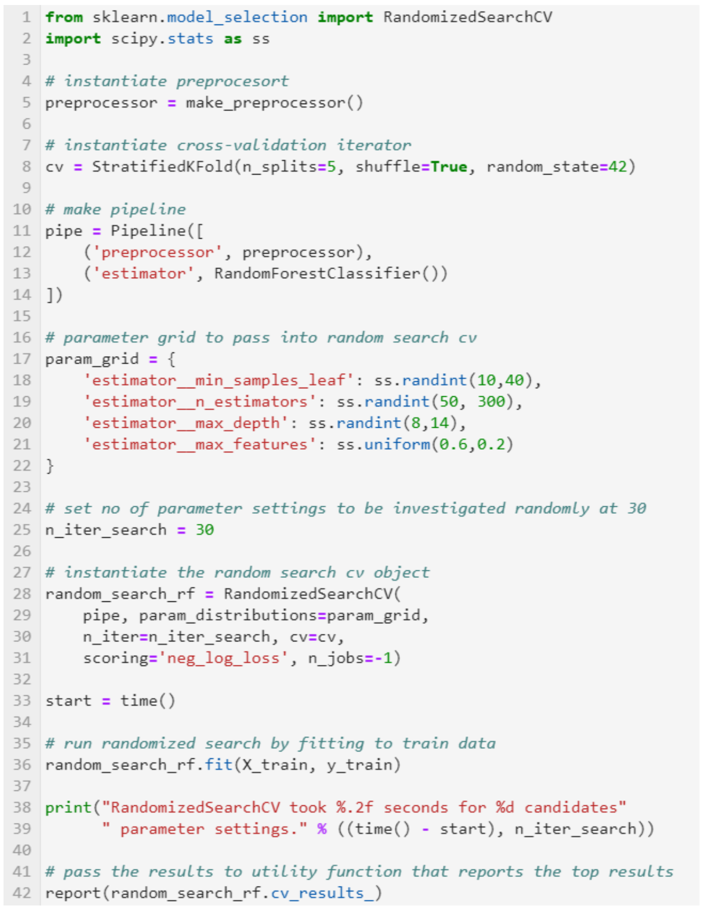
[14] usεr11852, “geometric mean for binary classification doesn't use sensitivity of each class,” [Online]. Available: https://stats.stackexchange.com/q/461809. [Accessed 19 March 2020].

Appendix



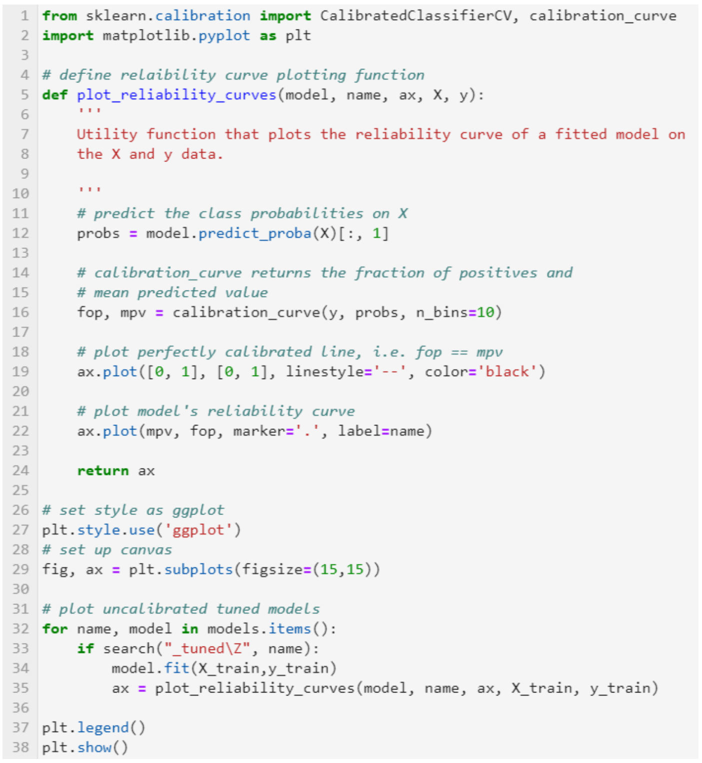
```python
from sklearn.model_selection import StratifiedKFold, validation_curve
from sklearn.pipeline import Pipeline

# instantiate preprocessor pipeline
preprocessor = make_preprocessor()

# instantiate cross-validation iterator
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# generate a range of parameter values
param_range = [10, 50, 100, 200, 400, 800, 1600]

# make pipeline
pipe = Pipeline([
    ('preprocessor', preprocessor),
    ('estimator', RandomForestClassifier())
])

# generate average cv train and test scores for each parameter
# value in the param_range
train_scores, test_scores = validation_curve(
    pipe, X_train, y_train, param_name="estimator__n_estimators",
    param_range=param_range,scoring="neg_log_loss",
    cv=cv, n_jobs=-1)

# pass scores to a utility function that plots the curve
plot_validation_curve(train_scores, test_scores,
                      'RF Validation Curve', 'n_estimators')
```

Figure A-1 – Plotting of the validation curve for RandomForestClassifier’s n_estimators parameter over a range of values with a 5-fold stratified cross-validation strategy



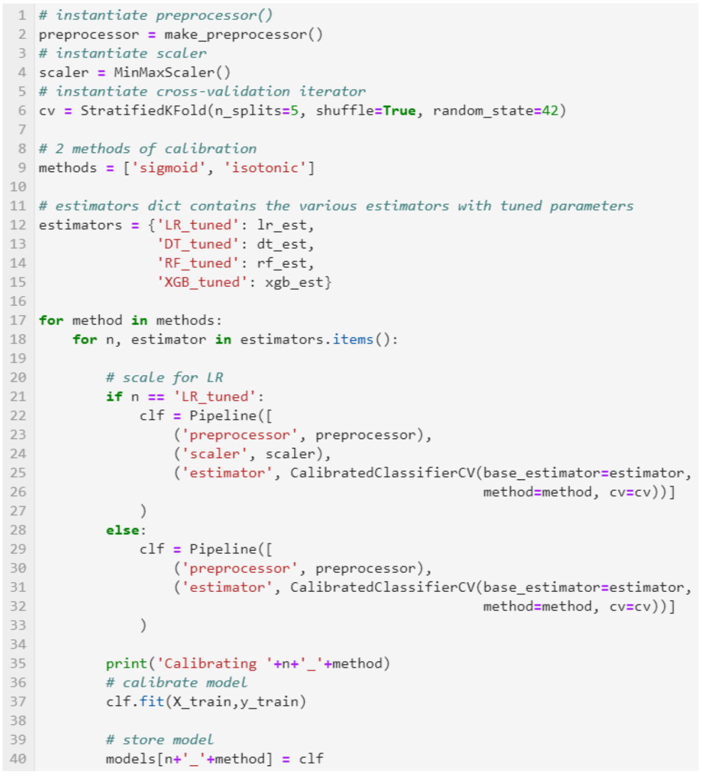
```python
from sklearn.model_selection import RandomizedSearchCV
import scipy.stats as ss

# instantiate preprocesort
preprocessor = make_preprocessor()

# instantiate cross-validation iterator
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# make pipeline
pipe = Pipeline([
    ('preprocessor', preprocessor),
    ('estimator', RandomForestClassifier())
])

# parameter grid to pass into random search cv
param_grid = {
    'estimator__min_samples_leaf': ss.randint(10,40),
    'estimator__n_estimators': ss.randint(50, 300),
    'estimator__max_depth': ss.randint(8,14),
    'estimator__max_features': ss.uniform(0.6,0.2)
}

# set no of parameter settings to be investigated randomly at 30
n_iter_search = 30

# instantiate the random search cv object
random_search_rf = RandomizedSearchCV(
    pipe, param_distributions=param_grid,
    n_iter=n_iter_search, cv=cv,
    scoring='neg_log_loss', n_jobs=-1)

start = time()

# run randomized search by fitting to train data
random_search_rf.fit(X_train, y_train)

print("RandomizedSearchCV took %.2f seconds for %d candidates"
      " parameter settings." % ((time() - start), n_iter_search))

# pass the results to utility function that reports the top results
report(random_search_rf.cv_results_)
```

Figure A-2 – Randomised search across a range of values for multiple parameters.



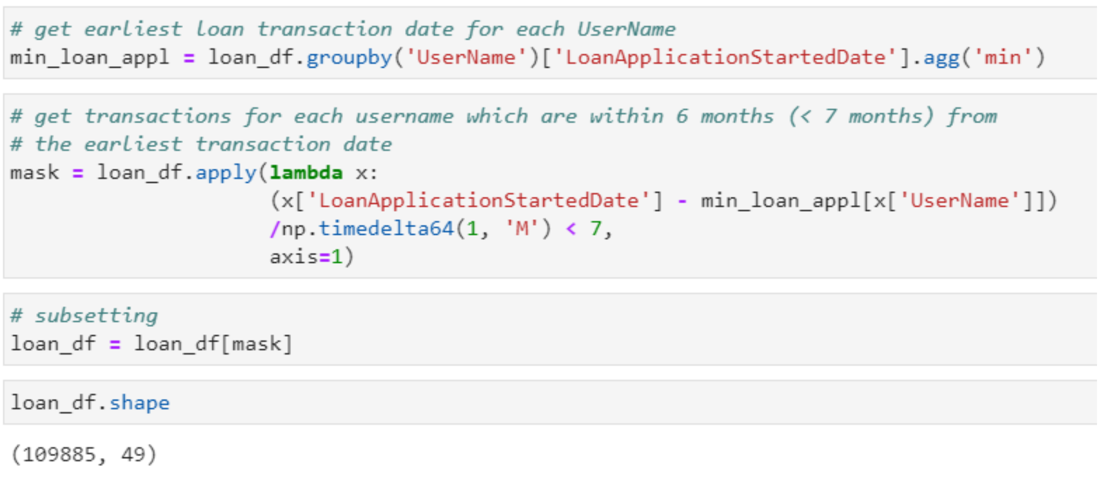
```python
from sklearn.calibration import CalibratedClassifierCV, calibration_curve
import matplotlib.pyplot as plt

# define relaibility curve plotting function
def plot_reliability_curves(model, name, ax, X, y):
    '''
    Utility function that plots the reliability curve of a fitted model on
    the X and y data.

    '''
    # predict the class probabilities on X
    probs = model.predict_proba(X)[:, 1]

    # calibration_curve returns the fraction of positives and
    # mean predicted value
    fop, mpv = calibration_curve(y, probs, n_bins=10)

    # plot perfectly calibrated line, i.e. fop == mpv
    ax.plot([0, 1], [0, 1], linestyle='--', color='black')

    # plot model's reliability curve
    ax.plot(mpv, fop, marker='.', label=name)

    return ax

# set style as ggplot
plt.style.use('ggplot')
# set up canvas
fig, ax = plt.subplots(figsize=(15,15))

# plot uncalibrated tuned models
for name, model in models.items():
    if search("_tuned\Z", name):
        model.fit(X_train,y_train)
        ax = plot_reliability_curves(model, name, ax, X_train, y_train)

plt.legend()
plt.show()
```

Figure A-3 – Plotting the reliability curves of the uncalibrated models.



```python
# instantiate preprocessor()
preprocessor = make_preprocessor()
# instantiate scaler
scaler = MinMaxScaler()
# instantiate cross-validation iterator
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# 2 methods of calibration
methods = ['sigmoid', 'isotonic']

# estimators dict contains the various estimators with tuned parameters
estimators = {'LR_tuned': lr_est,
              'DT_tuned': dt_est,
              'RF_tuned': rf_est,
              'XGB_tuned': xgb_est}

for method in methods:
    for n, estimator in estimators.items():

        # scale for LR
        if n == 'LR_tuned':
            clf = Pipeline([
                ('preprocessor', preprocessor),
                ('scaler', scaler),
                ('estimator', CalibratedClassifierCV(base_estimator=estimator,
                                                     method=method, cv=cv))]
            )
        else:
            clf = Pipeline([
                ('preprocessor', preprocessor),
                ('estimator', CalibratedClassifierCV(base_estimator=estimator,
                                                     method=method, cv=cv))]
            )

        print('Calibrating '+n+'_'+method)
        # calibrate model
        clf.fit(X_train,y_train)

        # store model
        models[n+'_'+method] = clf
```

Figure A-4 – Calibration of the tuned models with either the isotonic regressor or sigmoid (Platt's) regressor.



```
# get earliest loan transaction date for each UserName
min_loan_appl = loan_df.groupby('UserName')['LoanApplicationStartedDate'].agg('min')
```

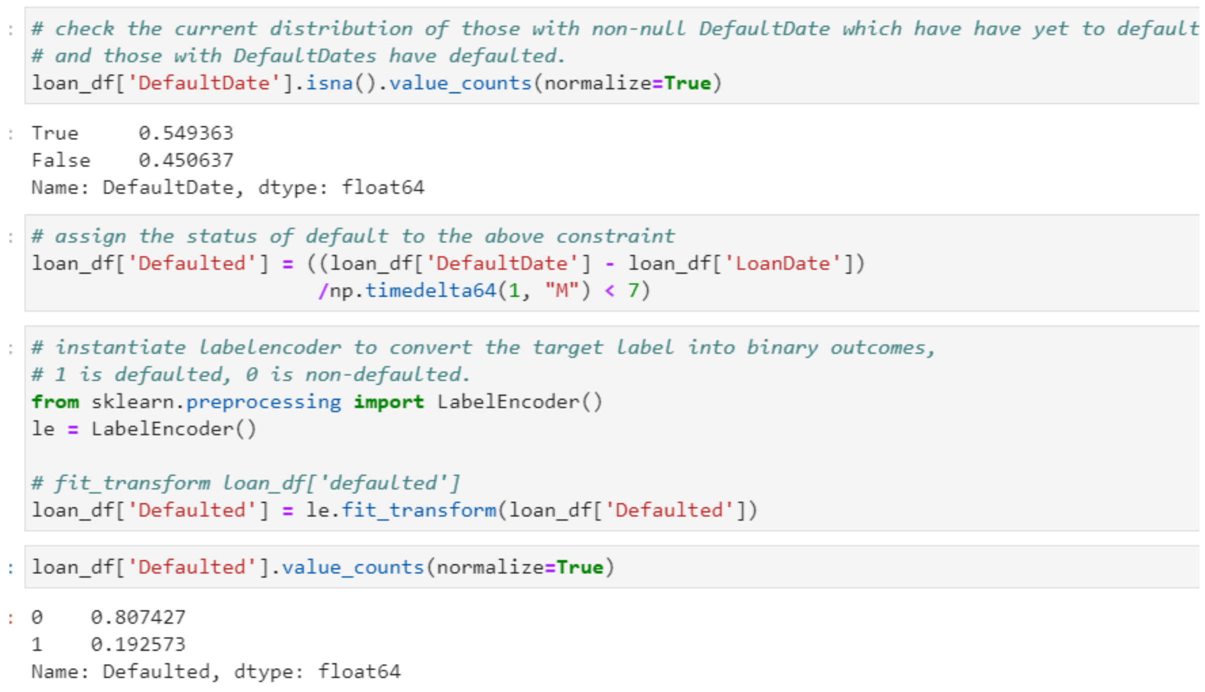
```
# get transactions for each username which are within 6 months (< 7 months) from
# the earliest transaction date
mask = loan_df.apply(lambda x:
                     (x['LoanApplicationStartedDate'] - min_loan_appl[x['UserName']])
                     /np.timedelta64(1, 'M') < 7,
                     axis=1)
```

```
# subsetting
loan_df = loan_df[mask]
```

```
loan_df.shape
```

```
(109885, 49)
```

Figure A-5 – Subsetting to thin-file/no-file clients and applications.



```
: # check the current distribution of those with non-null DefaultDate which have have yet to default
  # and those with DefaultDates have defaulted.
  loan_df['DefaultDate'].isna().value_counts(normalize=True)
```

```
: True     0.549363
  False    0.450637
  Name: DefaultDate, dtype: float64
```

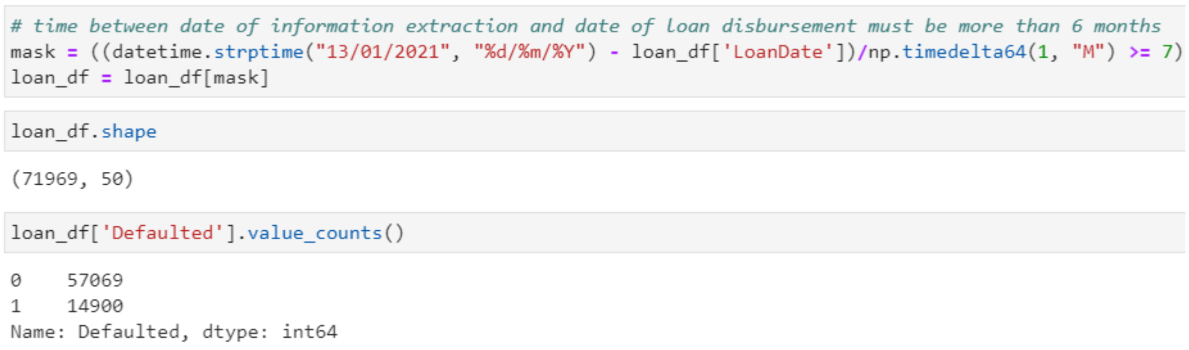
```
: # assign the status of default to the above constraint
  loan_df['Defaulted'] = ((loan_df['DefaultDate'] - loan_df['LoanDate'])
                          /np.timedelta64(1, "M") < 7)
```

```
: # instantiate labelencoder to convert the target label into binary outcomes,
  # 1 is defaulted, 0 is non-defaulted.
  from sklearn.preprocessing import LabelEncoder()
  le = LabelEncoder()

  # fit_transform loan_df['defaulted']
  loan_df['Defaulted'] = le.fit_transform(loan_df['Defaulted'])
```

```
: loan_df['Defaulted'].value_counts(normalize=True)
```

```
: 0    0.807427
  1    0.192573
  Name: Defaulted, dtype: float64
```

Figure A-6 – Target label construction



```
# time between date of information extraction and date of loan disbursement must be more than 6 months
mask = ((datetime.strptime("13/01/2021", "%d/%m/%Y") - loan_df['LoanDate'])/np.timedelta64(1, "M") >= 7)
loan_df = loan_df[mask]
```

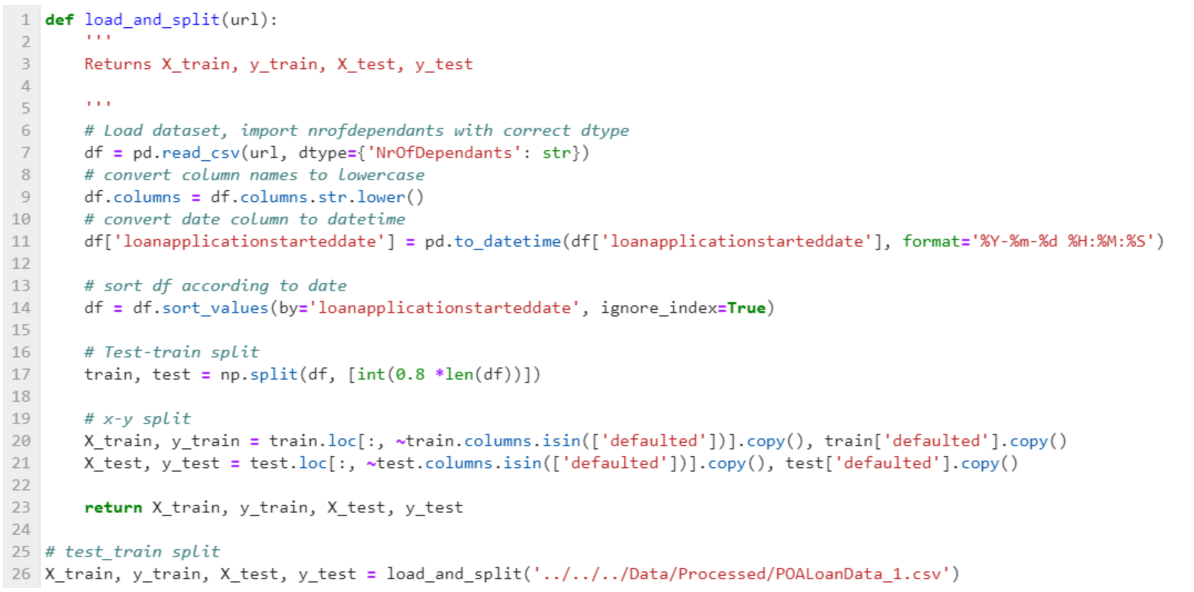
```
loan_df.shape
```

```
(71969, 50)
```

```
loan_df['Defaulted'].value_counts()
```

```
0    57069
1    14900
Name: Defaulted, dtype: int64
```

Figure A-7 – Further subsetting to ensure that the loans have more than 76 months of runway for us to be sure of their eventual default status.



```
def load_and_split(url):
    '''
    Returns X_train, y_train, X_test, y_test

    '''
    # Load dataset, import nrofdependants with correct dtype
    df = pd.read_csv(url, dtype={'NrOfDependants': str})
    # convert column names to lowercase
    df.columns = df.columns.str.lower()
    # convert date column to datetime
    df['loanapplicationstarteddate'] = pd.to_datetime(df['loanapplicationstarteddate'], format='%Y-%m-%d %H:%M:%S')

    # sort df according to date
    df = df.sort_values(by='loanapplicationstarteddate', ignore_index=True)

    # Test-train split
    train, test = np.split(df, [int(0.8 *len(df))])

    # x-y split
    X_train, y_train = train.loc[:, ~train.columns.isin(['defaulted'])].copy(), train['defaulted'].copy()
    X_test, y_test = test.loc[:, ~test.columns.isin(['defaulted'])].copy(), test['defaulted'].copy()

    return X_train, y_train, X_test, y_test

# test_train split
X_train, y_train, X_test, y_test = load_and_split('../../../Data/Processed/POALoanData_1.csv')
```

Figure A-8 – Custom function for train-test split

```python
# get pandas series of variables which have missing (NaN) values
# and the corresponding quantity
miss = X_train.isna().sum()[X_train.isna().sum() > 0].sort_values()

plt.style.use('ggplot')
# set canvas
_, ax = plt.subplots()
# plot barh chart
ax.barh(miss.index, miss.values)
# set log scale
ax.set_xscale('log')

for i, v in enumerate(miss):
    ax.text(v, i-0.1, str(v), color='grey')
```

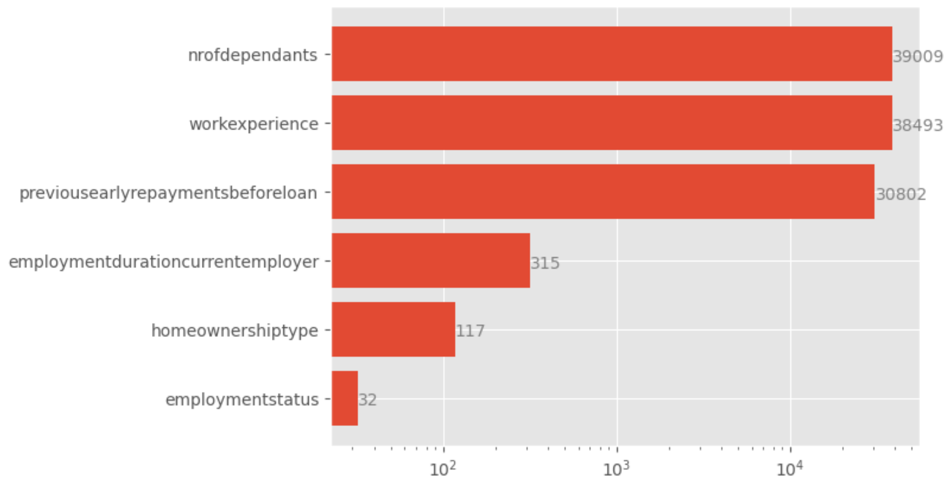
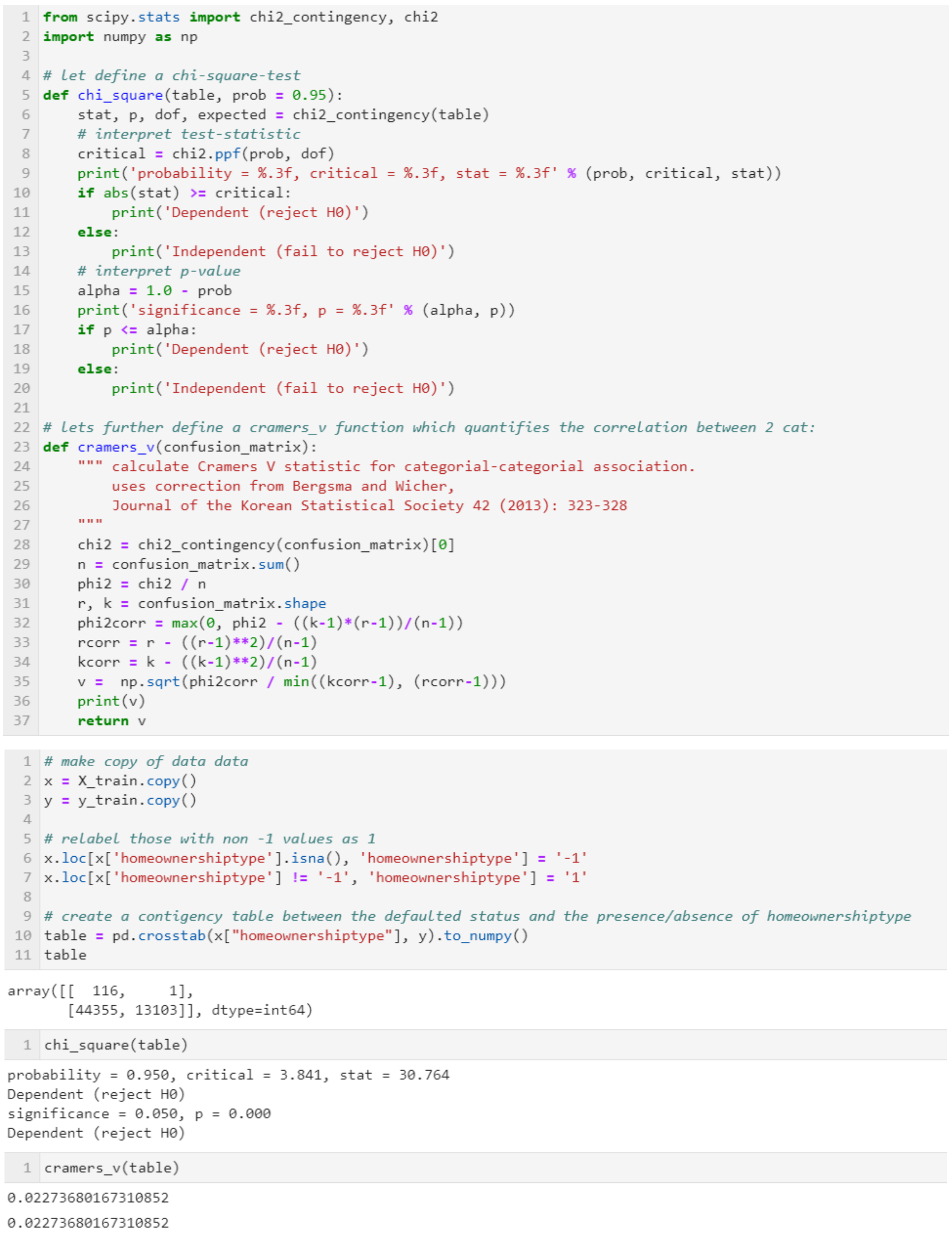


Figure A-9 – Investigating missing values.

```python
# dictionary of defined values for the categorical variables
dtype_dic = {
        'gender' : list(range(0,3)),
        'homeownershiptype': list(range(0,11)),
        'languagecode': [1, 2, 3, 4, 6, 9],
        'maritalstatus': list(range(1, 6)),
        'country' : ['EE', 'FI', 'ES','SK'],
        'employmentstatus' : list(range(1, 7)),
        'occupationarea' : list(range(0, 20)),
        'nrofdependants' : ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '10Plus'],
        'employmentdurationcurrentemployer' : ['Retiree', 'Other', 'TrialPeriod','UpTo1Year', 'UpTo2Years',
                                               'UpTo3Years', 'UpTo4Years', 'UpTo5Years', 'MoreThan5Years'],
        'education' : list(range(1,6)),
        'workexperience' : ['LessThan2Years','2To5Years','5To10Years','10To15Years','15To25Years','MoreThan25Years']
    }

# categorical variable list
cat_var = ['gender', 'homeownershiptype','languagecode', 'maritalstatus', 'nrofdependants', 'country',
           'education', 'employmentdurationcurrentemployer', 'employmentstatus','occupationarea', 'workexperience']

# for each var in cat_var
for var in cat_var:
    # get values which are not in the defined values and is not nan
    mask = ~(X_train[var].isin(dtype_dic[var]) | X_train[var].isna())
    counts = len(X_train[var][mask])

    if counts != 0:
        print(f'{var}, Counts: {counts}, Undefined values: {X_train[var][mask].unique()}')
```

```
homeownershiptype, Counts: 2, Undefined values: [-1.]
languagecode, Counts: 10, Undefined values: [ 5 10 13  7 21 22 15]
maritalstatus, Counts: 38488, Undefined values: [-1]
education, Counts: 2, Undefined values: [-1]
employmentstatus, Counts: 38510, Undefined values: [ 0. -1.]
occupationarea, Counts: 38519, Undefined values: [-1]
```

Figure A-10 – Investigating undefined values



```python
from scipy.stats import chi2_contingency, chi2
import numpy as np

# let define a chi-square-test
def chi_square(table, prob = 0.95):
    stat, p, dof, expected = chi2_contingency(table)
    # interpret test-statistic
    critical = chi2.ppf(prob, dof)
    print('probability = %.3f, critical = %.3f, stat = %.3f' % (prob, critical, stat))
    if abs(stat) >= critical:
        print('Dependent (reject H0)')
    else:
        print('Independent (fail to reject H0)')
    # interpret p-value
    alpha = 1.0 - prob
    print('significance = %.3f, p = %.3f' % (alpha, p))
    if p <= alpha:
        print('Dependent (reject H0)')
    else:
        print('Independent (fail to reject H0)')

# lets further define a cramers_v function which quantifies the correlation between 2 cat:
def cramers_v(confusion_matrix):
    """ calculate Cramers V statistic for categorial-categorial association.
        uses correction from Bergsma and Wicher,
        Journal of the Korean Statistical Society 42 (2013): 323-328
    """
    chi2 = chi2_contingency(confusion_matrix)[0]
    n = confusion_matrix.sum()
    phi2 = chi2 / n
    r, k = confusion_matrix.shape
    phi2corr = max(0, phi2 - ((k-1)*(r-1))/(n-1))
    rcorr = r - ((r-1)**2)/(n-1)
    kcorr = k - ((k-1)**2)/(n-1)
    v =  np.sqrt(phi2corr / min((kcorr-1), (rcorr-1)))
    print(v)
    return v
```

```python
# make copy of data data
x = X_train.copy()
y = y_train.copy()

# relabel those with non -1 values as 1
x.loc[x['homeownershiptype'].isna(), 'homeownershiptype'] = '-1'
x.loc[x['homeownershiptype'] != '-1', 'homeownershiptype'] = '1'

# create a contigency table between the defaulted status and the presence/absence of homeownershiptype
table = pd.crosstab(x["homeownershiptype"], y).to_numpy()
table
```

```
array([[  116,     1],
       [44355, 13103]], dtype=int64)
```

```python
chi_square(table)
```

```
probability = 0.950, critical = 3.841, stat = 30.764
Dependent (reject H0)
significance = 0.050, p = 0.000
Dependent (reject H0)
```

```python
cramers_v(table)
```

```
0.02273680167310852
0.02273680167310852
```

Figure A-11 – Chi-square test and Cramer's V test for independence between the missing values and the dependent values.

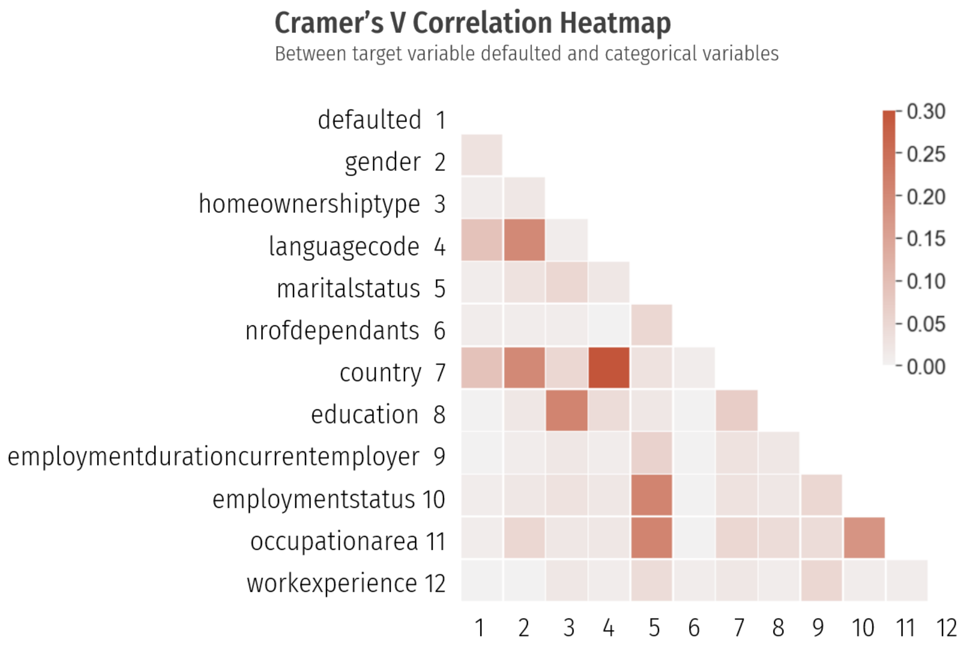
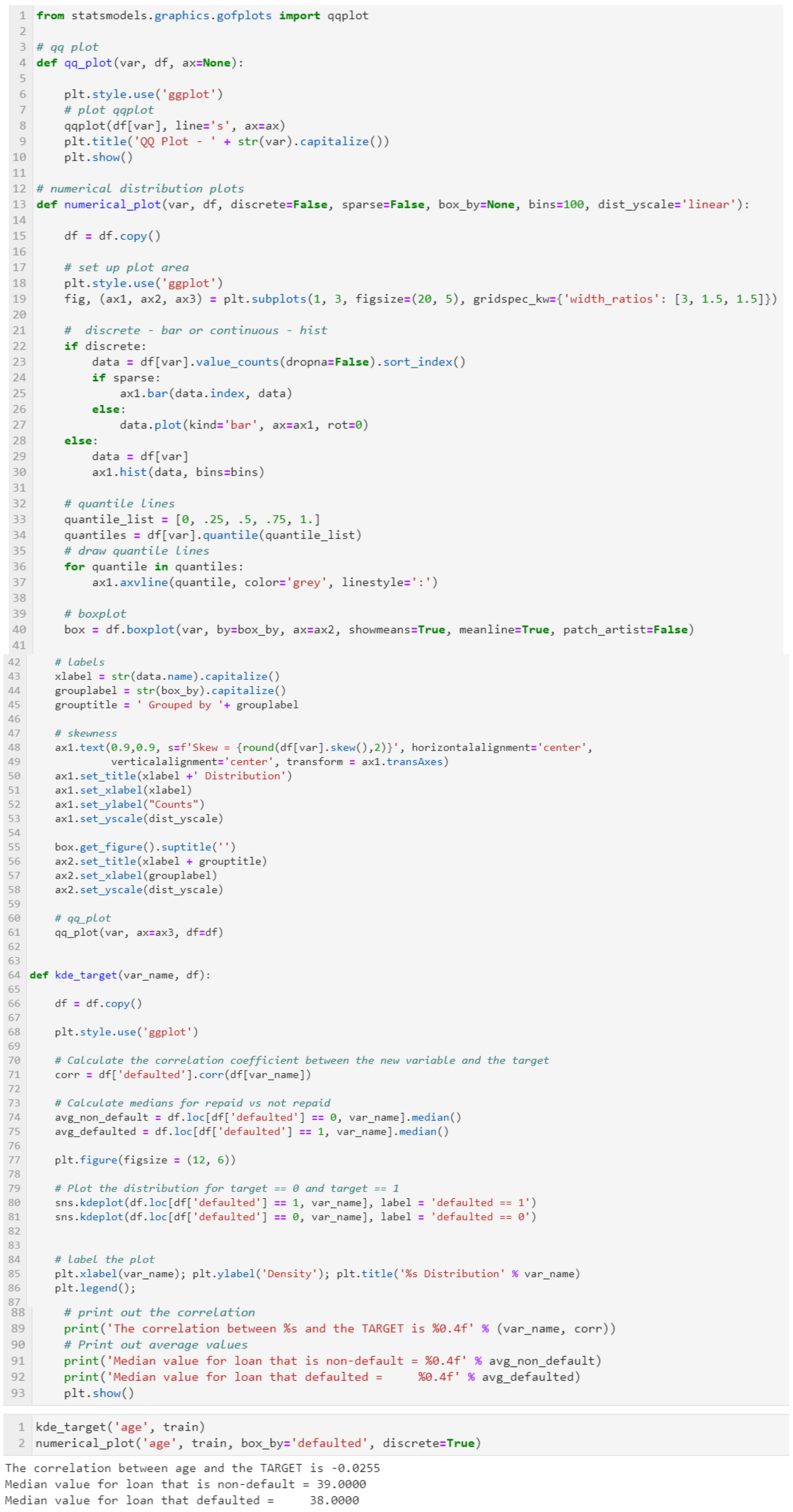


Figure A-12 – Cramer's V correlation heatmap. Poor correlation of the dependent variable defaulted to the other variables.

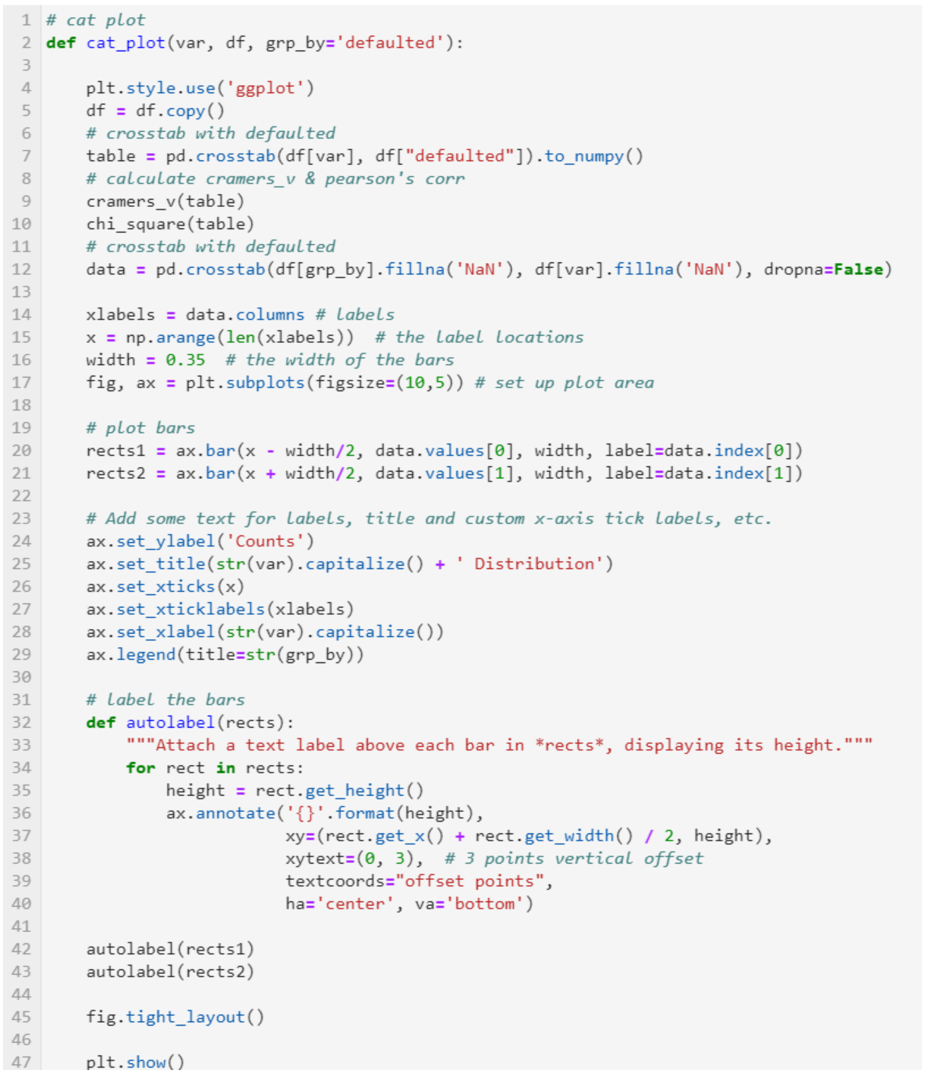


```python
from statsmodels.graphics.gofplots import qqplot

# qq plot
def qq_plot(var, df, ax=None):

    plt.style.use('ggplot')
    # plot qqplot
    qqplot(df[var], line='s', ax=ax)
    plt.title('QQ Plot - ' + str(var).capitalize())
    plt.show()

# numerical distribution plots
def numerical_plot(var, df, discrete=False, sparse=False, box_by=None, bins=100, dist_yscale='linear'):

    df = df.copy()

    # set up plot area
    plt.style.use('ggplot')
    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20, 5), gridspec_kw={'width_ratios': [3, 1.5, 1.5]})

    #  discrete - bar or continuous - hist
    if discrete:
        data = df[var].value_counts(dropna=False).sort_index()
        if sparse:
            ax1.bar(data.index, data)
        else:
            data.plot(kind='bar', ax=ax1, rot=0)
    else:
        data = df[var]
        ax1.hist(data, bins=bins)

    # quantile lines
    quantile_list = [0, .25, .5, .75, 1.]
    quantiles = df[var].quantile(quantile_list)
    # draw quantile lines
    for quantile in quantiles:
        ax1.axvline(quantile, color='grey', linestyle=':')

    # boxplot
    box = df.boxplot(var, by=box_by, ax=ax2, showmeans=True, meanline=True, patch_artist=False)

    # labels
    xlabel = str(data.name).capitalize()
    grouplabel = str(box_by).capitalize()
    grouptitle = ' Grouped by '+ grouplabel

    # skewness
    ax1.text(0.9,0.9, s=f'Skew = {round(df[var].skew(),2)}', horizontalalignment='center',
             verticalalignment='center', transform = ax1.transAxes)
    ax1.set_title(xlabel +' Distribution')
    ax1.set_xlabel(xlabel)
    ax1.set_ylabel("Counts")
    ax1.set_yscale(dist_yscale)

    box.get_figure().suptitle('')
    ax2.set_title(xlabel + grouptitle)
    ax2.set_xlabel(grouplabel)
    ax2.set_yscale(dist_yscale)

    # qq_plot
    qq_plot(var, ax=ax3, df=df)


def kde_target(var_name, df):

    df = df.copy()

    plt.style.use('ggplot')

    # Calculate the correlation coefficient between the new variable and the target
    corr = df['defaulted'].corr(df[var_name])

    # Calculate medians for repaid vs not repaid
    avg_non_default = df.loc[df['defaulted'] == 0, var_name].median()
    avg_defaulted = df.loc[df['defaulted'] == 1, var_name].median()

    plt.figure(figsize = (12, 6))

    # Plot the distribution for target == 0 and target == 1
    sns.kdeplot(df.loc[df['defaulted'] == 1, var_name], label = 'defaulted == 1')
    sns.kdeplot(df.loc[df['defaulted'] == 0, var_name], label = 'defaulted == 0')


    # label the plot
    plt.xlabel(var_name); plt.ylabel('Density'); plt.title('%s Distribution' % var_name)
    plt.legend();

    # print out the correlation
    print('The correlation between %s and the TARGET is %0.4f' % (var_name, corr))
    # Print out average values
    print('Median value for loan that is non-default = %0.4f' % avg_non_default)
    print('Median value for loan that defaulted =     %0.4f' % avg_defaulted)
    plt.show()
```

```python
kde_target('age', train)
numerical_plot('age', train, box_by='defaulted', discrete=True)
```

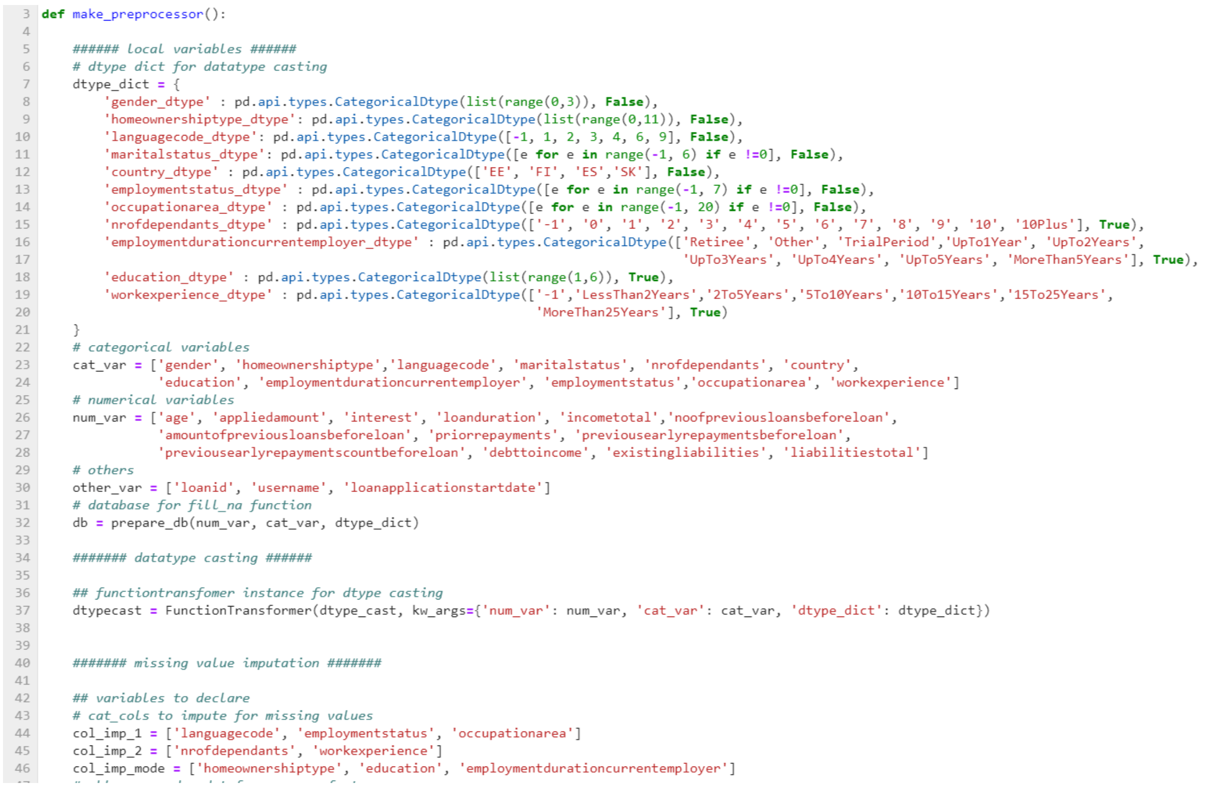
```
The correlation between age and the TARGET is -0.0255
Median value for loan that is non-default = 39.0000
Median value for loan that defaulted =     38.0000
```

Figure A-13 – Functions to plot the KDE, boxplots, Q-Q plots and distribution for numerical variables

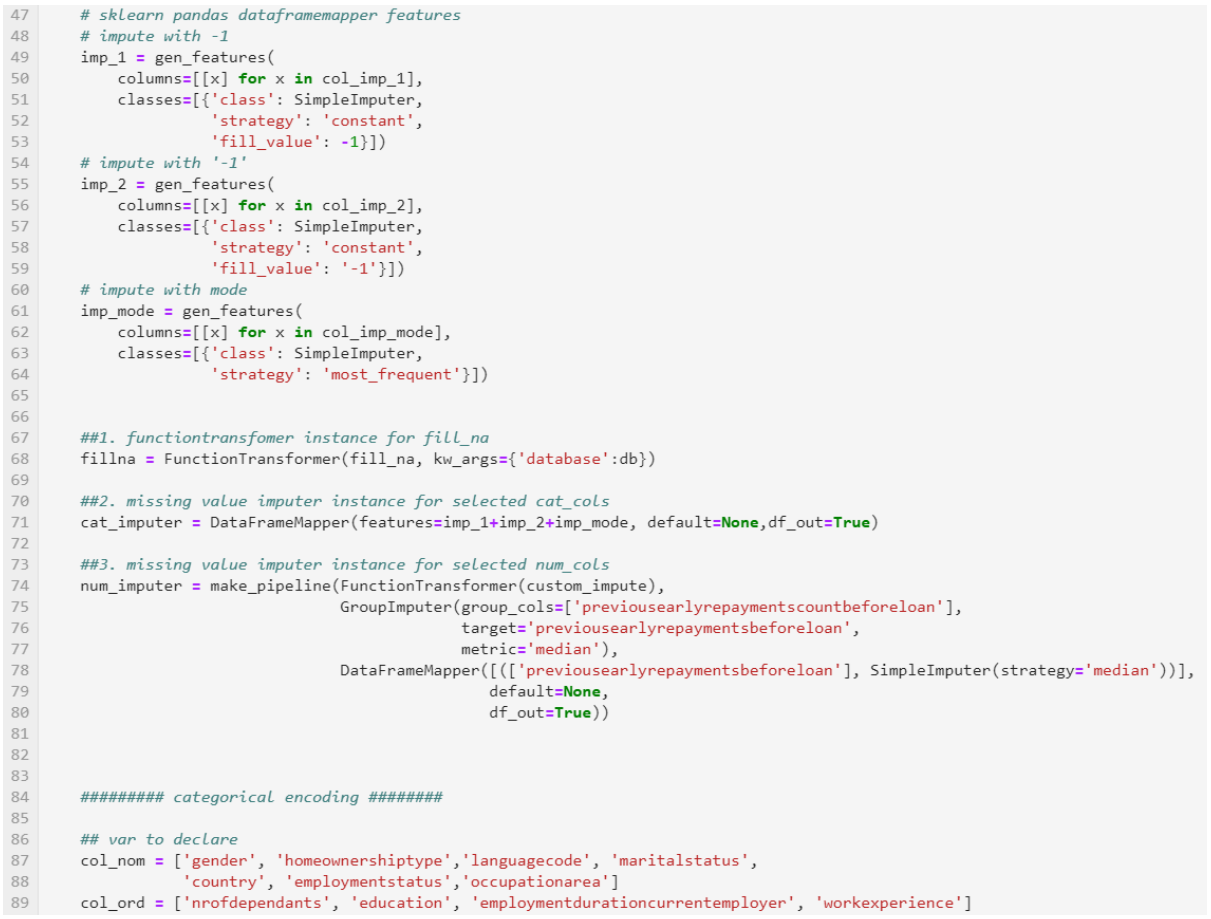


```python
# cat plot
def cat_plot(var, df, grp_by='defaulted'):

    plt.style.use('ggplot')
    df = df.copy()
    # crosstab with defaulted
    table = pd.crosstab(df[var], df["defaulted"]).to_numpy()
    # calculate cramers_v & pearson's corr
    cramers_v(table)
    chi_square(table)
    # crosstab with defaulted
    data = pd.crosstab(df[grp_by].fillna('NaN'), df[var].fillna('NaN'), dropna=False)

    xlabels = data.columns # labels
    x = np.arange(len(xlabels))  # the label locations
    width = 0.35  # the width of the bars
    fig, ax = plt.subplots(figsize=(10,5)) # set up plot area

    # plot bars
    rects1 = ax.bar(x - width/2, data.values[0], width, label=data.index[0])
    rects2 = ax.bar(x + width/2, data.values[1], width, label=data.index[1])

    # Add some text for labels, title and custom x-axis tick labels, etc.
    ax.set_ylabel('Counts')
    ax.set_title(str(var).capitalize() + ' Distribution')
    ax.set_xticks(x)
    ax.set_xticklabels(xlabels)
    ax.set_xlabel(str(var).capitalize())
    ax.legend(title=str(grp_by))

    # label the bars
    def autolabel(rects):
        """Attach a text label above each bar in *rects*, displaying its height."""
        for rect in rects:
            height = rect.get_height()
            ax.annotate('{}'.format(height),
                        xy=(rect.get_x() + rect.get_width() / 2, height),
                        xytext=(0, 3),  # 3 points vertical offset
                        textcoords="offset points",
                        ha='center', va='bottom')

    autolabel(rects1)
    autolabel(rects2)

    fig.tight_layout()

    plt.show()
```

Figure A-14 – Plotting function for the bar plot for categorical variables

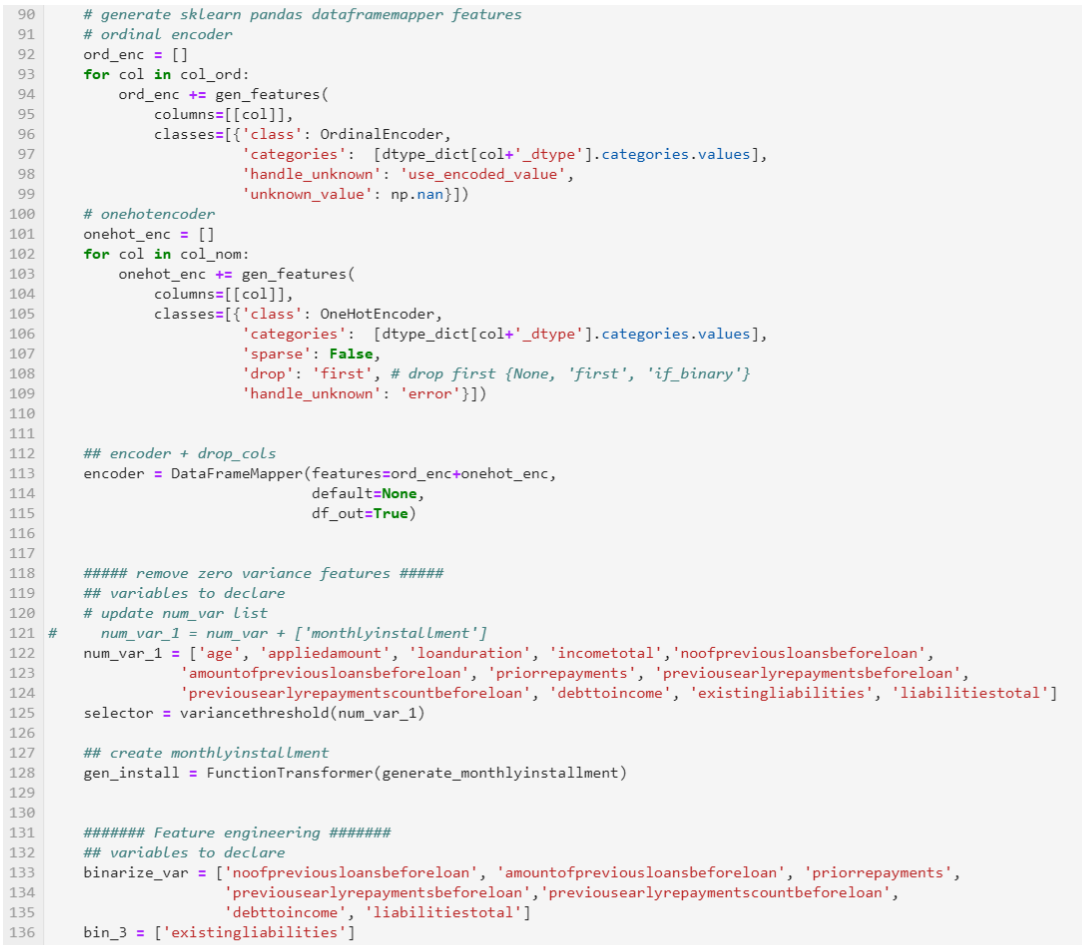


```python
def make_preprocessor():

    ###### local variables ######
    # dtype dict for datatype casting
    dtype_dict = {
        'gender_dtype' : pd.api.types.CategoricalDtype(list(range(0,3)), False),
        'homeownershiptype_dtype': pd.api.types.CategoricalDtype(list(range(0,11)), False),
        'languagecode_dtype': pd.api.types.CategoricalDtype([-1, 1, 2, 3, 4, 6, 9], False),
        'maritalstatus_dtype': pd.api.types.CategoricalDtype([e for e in range(-1, 6) if e !=0], False),
        'country_dtype' : pd.api.types.CategoricalDtype(['EE', 'FI', 'ES','SK'], False),
        'employmentstatus_dtype' : pd.api.types.CategoricalDtype([e for e in range(-1, 7) if e !=0], False),
        'occupationarea_dtype' : pd.api.types.CategoricalDtype([e for e in range(-1, 20) if e !=0], False),
        'nrofdependants_dtype' : pd.api.types.CategoricalDtype(['-1', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '10Plus'], True),
        'employmentdurationcurrentemployer_dtype' : pd.api.types.CategoricalDtype(['Retiree', 'Other', 'TrialPeriod','UpTo1Year', 'UpTo2Years',
                                                                                   'UpTo3Years', 'UpTo4Years', 'UpTo5Years', 'MoreThan5Years'], True),
        'education_dtype' : pd.api.types.CategoricalDtype(list(range(1,6)), True),
        'workexperience_dtype' : pd.api.types.CategoricalDtype(['-1','LessThan2Years','2To5Years','5To10Years','10To15Years','15To25Years',
                                                                 'MoreThan25Years'], True)
    }
    # categorical variables
    cat_var = ['gender', 'homeownershiptype','languagecode', 'maritalstatus', 'nrofdependants', 'country',
               'education', 'employmentdurationcurrentemployer', 'employmentstatus','occupationarea', 'workexperience']
    # numerical variables
    num_var = ['age', 'appliedamount', 'interest', 'loanduration', 'incometotal','noofpreviousloansbeforeloan',
               'amountofpreviousloansbeforeloan', 'priorrepayments', 'previousearlyrepaymentsbeforeloan',
               'previousearlyrepaymentscountbeforeloan', 'debttoincome', 'existingliabilities', 'liabilitiestotal']
    # others
    other_var = ['loanid', 'username', 'loanapplicationstartdate']
    # database for fill_na function
    db = prepare_db(num_var, cat_var, dtype_dict)

    ####### datatype casting ######

    ## functiontransfomer instance for dtype casting
    dtypecast = FunctionTransformer(dtype_cast, kw_args={'num_var': num_var, 'cat_var': cat_var, 'dtype_dict': dtype_dict})


    ####### missing value imputation #######

    ## variables to declare
    # cat_cols to impute for missing values
    col_imp_1 = ['languagecode', 'employmentstatus', 'occupationarea']
    col_imp_2 = ['nrofdependants', 'workexperience']
    col_imp_mode = ['homeownershiptype', 'education', 'employmentdurationcurrentemployer']
```

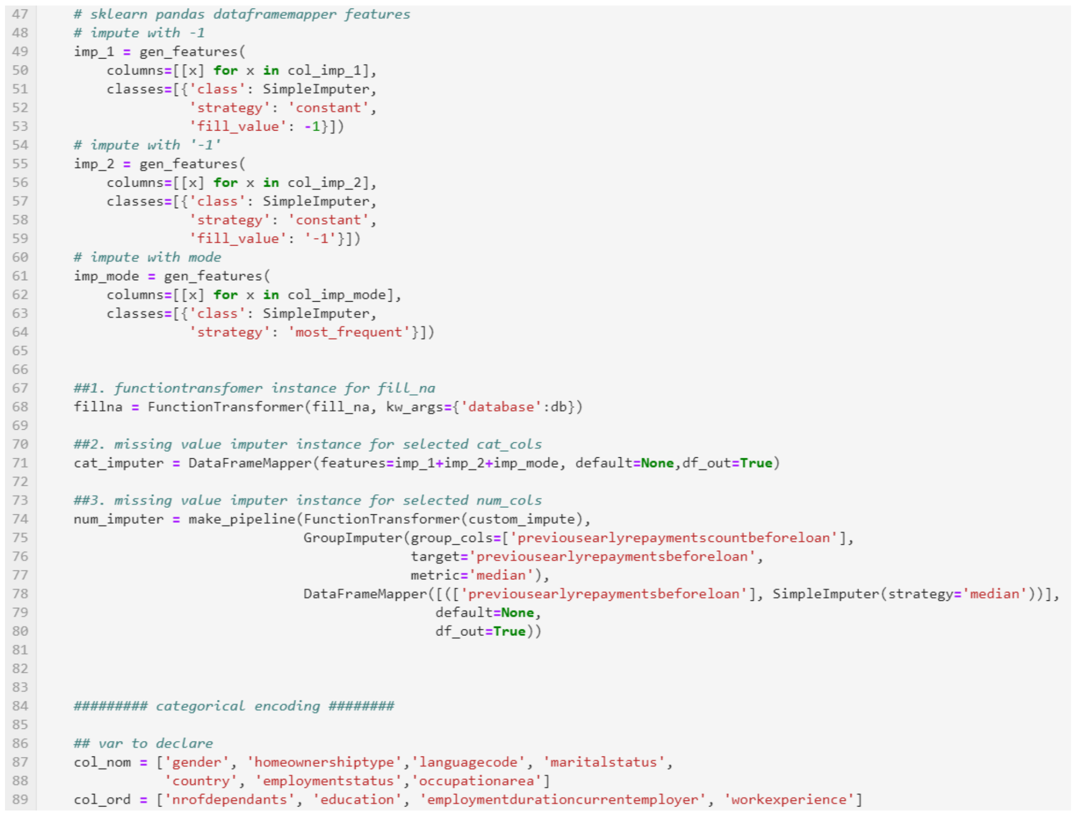


```python
    # sklearn pandas dataframemapper features
    # impute with -1
    imp_1 = gen_features(
        columns=[[x] for x in col_imp_1],
        classes=[{'class': SimpleImputer,
                  'strategy': 'constant',
                  'fill_value': -1}])
    # impute with '-1'
    imp_2 = gen_features(
        columns=[[x] for x in col_imp_2],
        classes=[{'class': SimpleImputer,
                  'strategy': 'constant',
                  'fill_value': '-1'}])
    # impute with mode
    imp_mode = gen_features(
        columns=[[x] for x in col_imp_mode],
        classes=[{'class': SimpleImputer,
                  'strategy': 'most_frequent'}])


    ##1. functiontransfomer instance for fill_na
    fillna = FunctionTransformer(fill_na, kw_args={'database':db})

    ##2. missing value imputer instance for selected cat_cols
    cat_imputer = DataFrameMapper(features=imp_1+imp_2+imp_mode, default=None,df_out=True)

    ##3. missing value imputer instance for selected num_cols
    num_imputer = make_pipeline(FunctionTransformer(custom_impute),
                                GroupImputer(group_cols=['previousearlyrepaymentscountbeforeloan'],
                                             target='previousearlyrepaymentsbeforeloan',
                                             metric='median'),
                                DataFrameMapper([(['previousearlyrepaymentsbeforeloan'], SimpleImputer(strategy='median'))],
                                                default=None,
                                                df_out=True))



    ######### categorical encoding ########

    ## var to declare
    col_nom = ['gender', 'homeownershiptype','languagecode', 'maritalstatus',
               'country', 'employmentstatus','occupationarea']
    col_ord = ['nrofdependants', 'education', 'employmentdurationcurrentemployer', 'workexperience']
```

Figure A-15 – Preprocessing pipeline (part 1)

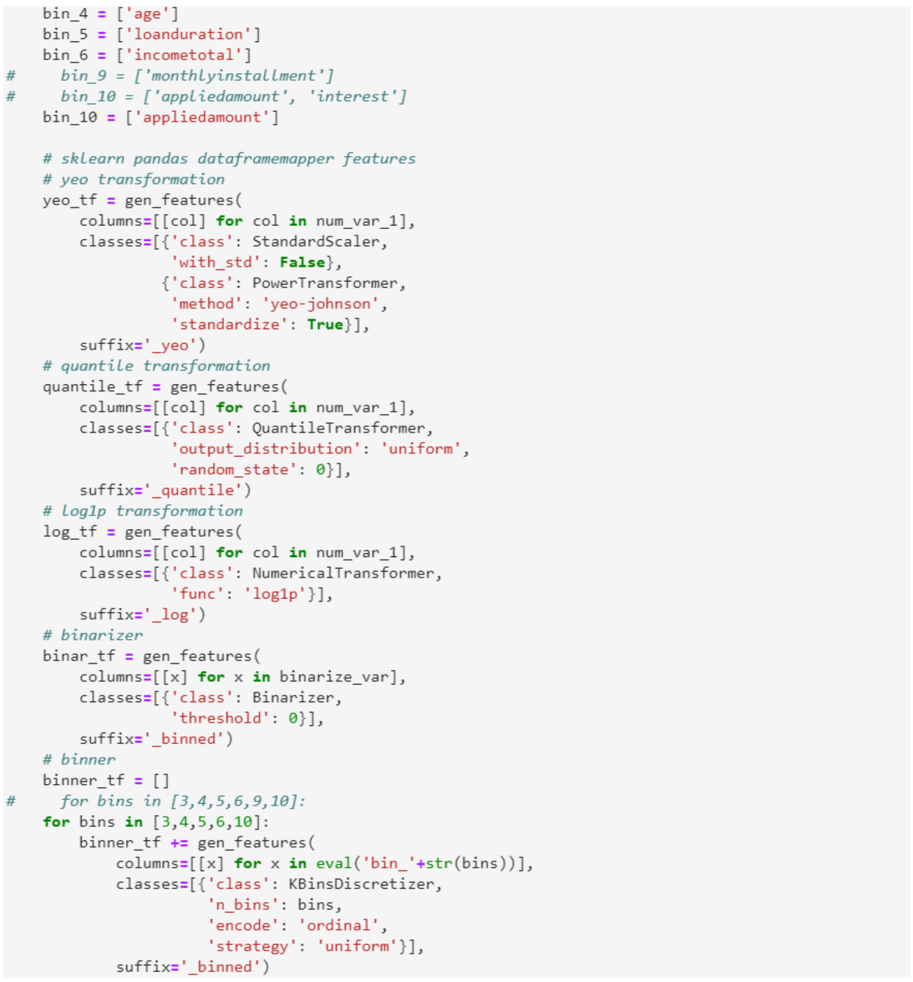


```python
    # generate sklearn pandas dataframemapper features
    # ordinal encoder
    ord_enc = []
    for col in col_ord:
        ord_enc += gen_features(
            columns=[[col]],
            classes=[{'class': OrdinalEncoder,
                      'categories':  [dtype_dict[col+'_dtype'].categories.values],
                      'handle_unknown': 'use_encoded_value',
                      'unknown_value': np.nan}])
    # onehotencoder
    onehot_enc = []
    for col in col_nom:
        onehot_enc += gen_features(
            columns=[[col]],
            classes=[{'class': OneHotEncoder,
                      'categories':  [dtype_dict[col+'_dtype'].categories.values],
                      'sparse': False,
                      'drop': 'first', # drop first {None, 'first', 'if_binary'}
                      'handle_unknown': 'error'}])


    ## encoder + drop_cols
    encoder = DataFrameMapper(features=ord_enc+onehot_enc,
                              default=None,
                              df_out=True)


    ##### remove zero variance features #####
    ## variables to declare
    # update num_var list
#     num_var_1 = num_var + ['monthlyinstallment']
    num_var_1 = ['age', 'appliedamount', 'loanduration', 'incometotal','noofpreviousloansbeforeloan',
                'amountofpreviousloansbeforeloan', 'priorrepayments', 'previousearlyrepaymentsbeforeloan',
                'previousearlyrepaymentscountbeforeloan', 'debttoincome', 'existingliabilities', 'liabilitiestotal']
    selector = variancethreshold(num_var_1)

    ## create monthlyinstallment
    gen_install = FunctionTransformer(generate_monthlyinstallment)


    ####### Feature engineering #######
    ## variables to declare
    binarize_var = ['noofpreviousloansbeforeloan', 'amountofpreviousloansbeforeloan', 'priorrepayments',
                    'previousearlyrepaymentsbeforeloan','previousearlyrepaymentscountbeforeloan',
                    'debttoincome', 'liabilitiestotal']
    bin_3 = ['existingliabilities']
```



```python
    # sklearn pandas dataframemapper features
    # impute with -1
    imp_1 = gen_features(
        columns=[[x] for x in col_imp_1],
        classes=[{'class': SimpleImputer,
                  'strategy': 'constant',
                  'fill_value': -1}])
    # impute with '-1'
    imp_2 = gen_features(
        columns=[[x] for x in col_imp_2],
        classes=[{'class': SimpleImputer,
                  'strategy': 'constant',
                  'fill_value': '-1'}])
    # impute with mode
    imp_mode = gen_features(
        columns=[[x] for x in col_imp_mode],
        classes=[{'class': SimpleImputer,
                  'strategy': 'most_frequent'}])


    ##1. functiontransfomer instance for fill_na
    fillna = FunctionTransformer(fill_na, kw_args={'database':db})

    ##2. missing value imputer instance for selected cat_cols
    cat_imputer = DataFrameMapper(features=imp_1+imp_2+imp_mode, default=None,df_out=True)

    ##3. missing value imputer instance for selected num_cols
    num_imputer = make_pipeline(FunctionTransformer(custom_impute),
                                GroupImputer(group_cols=['previousearlyrepaymentscountbeforeloan'],
                                             target='previousearlyrepaymentsbeforeloan',
                                             metric='median'),
                                DataFrameMapper([(['previousearlyrepaymentsbeforeloan'], SimpleImputer(strategy='median'))],
                                                default=None,
                                                df_out=True))



    ######## categorical encoding ########

    ## var to declare
    col_nom = ['gender', 'homeownershiptype','languagecode', 'maritalstatus',
               'country', 'employmentstatus','occupationarea']
    col_ord = ['nrofdependants', 'education', 'employmentdurationcurrentemployer', 'workexperience']
```

Figure A-16 – Preprocessing pipeline (part 2)


```
    bin_4 = ['age']
    bin_5 = ['loanduration']
    bin_6 = ['incometotal']
#     bin_9 = ['monthlyinstallment']
#     bin_10 = ['appliedamount', 'interest']
    bin_10 = ['appliedamount']

    # sklearn pandas dataframemapper features
    # yeo transformation
    yeo_tf = gen_features(
        columns=[[col] for col in num_var_1],
        classes=[{'class': StandardScaler,
                  'with_std': False},
                 {'class': PowerTransformer,
                  'method': 'yeo-johnson',
                  'standardize': True}],
        suffix='_yeo')
    # quantile transformation
    quantile_tf = gen_features(
        columns=[[col] for col in num_var_1],
        classes=[{'class': QuantileTransformer,
                  'output_distribution': 'uniform',
                  'random_state': 0}],
        suffix='_quantile')
    # log1p transformation
    log_tf = gen_features(
        columns=[[col] for col in num_var_1],
        classes=[{'class': NumericalTransformer,
                  'func': 'log1p'}],
        suffix='_log')
    # binarizer
    binar_tf = gen_features(
        columns=[[x] for x in binarize_var],
        classes=[{'class': Binarizer,
                  'threshold': 0}],
        suffix='_binned')
    # binner
    binner_tf = []
#     for bins in [3,4,5,6,9,10]:
    for bins in [3,4,5,6,10]:
        binner_tf += gen_features(
            columns=[[x] for x in eval('bin_'+str(bins))],
            classes=[{'class': KBinsDiscretizer,
                      'n_bins': bins,
                      'encode': 'ordinal',
                      'strategy': 'uniform'}],
            suffix='_binned')
```

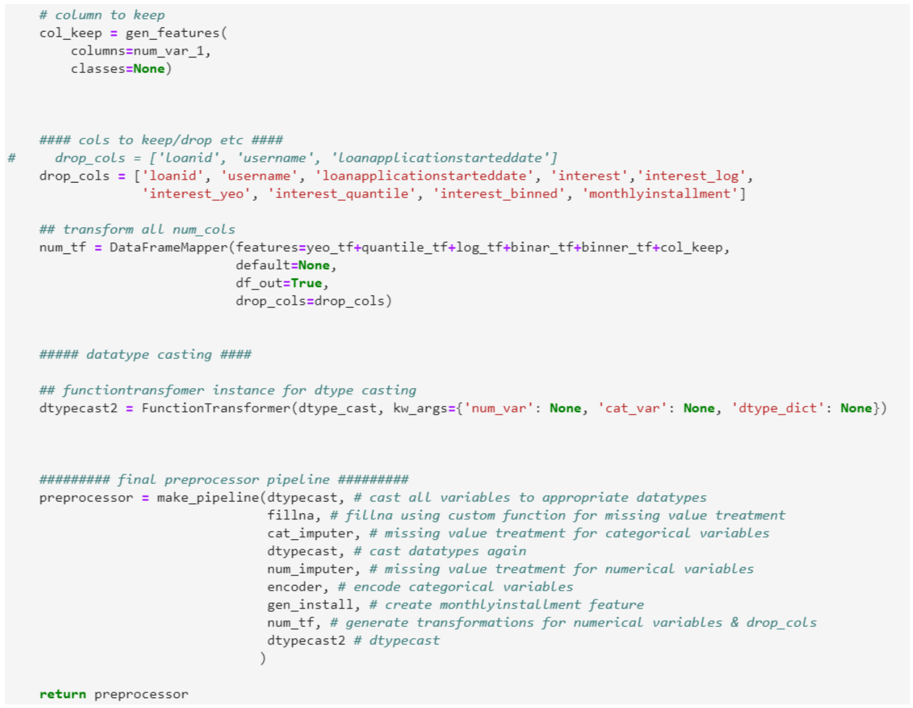

```
    # column to keep
    col_keep = gen_features(
        columns=num_var_1,
        classes=None)



    #### cols to keep/drop etc ####
#     drop_cols = ['loanid', 'username', 'loanapplicationstarteddate']
    drop_cols = ['loanid', 'username', 'loanapplicationstarteddate', 'interest','interest_log',
                 'interest_yeo', 'interest_quantile', 'interest_binned', 'monthlyinstallment']

    ## transform all num_cols
    num_tf = DataFrameMapper(features=yeo_tf+quantile_tf+log_tf+binar_tf+binner_tf+col_keep,
                             default=None,
                             df_out=True,
                             drop_cols=drop_cols)


    ##### datatype casting ####

    ## functiontransfomer instance for dtype casting
    dtypecast2 = FunctionTransformer(dtype_cast, kw_args={'num_var': None, 'cat_var': None, 'dtype_dict': None})



    ######### final preprocessor pipeline #########
    preprocessor = make_pipeline(dtypecast, # cast all variables to appropriate datatypes
                                 fillna, # fillna using custom function for missing value treatment
                                 cat_imputer, # missing value treatment for categorical variables
                                 dtypecast, # cast datatypes again
                                 num_imputer, # missing value treatment for numerical variables
                                 encoder, # encode categorical variables
                                 gen_install, # create monthlyinstallment feature
                                 num_tf, # generate transformations for numerical variables & drop_cols
                                 dtypecast2 # dtypecast
                                )

    return preprocessor
```

Figure A-17 – Preprocessing pipeline (part 3)

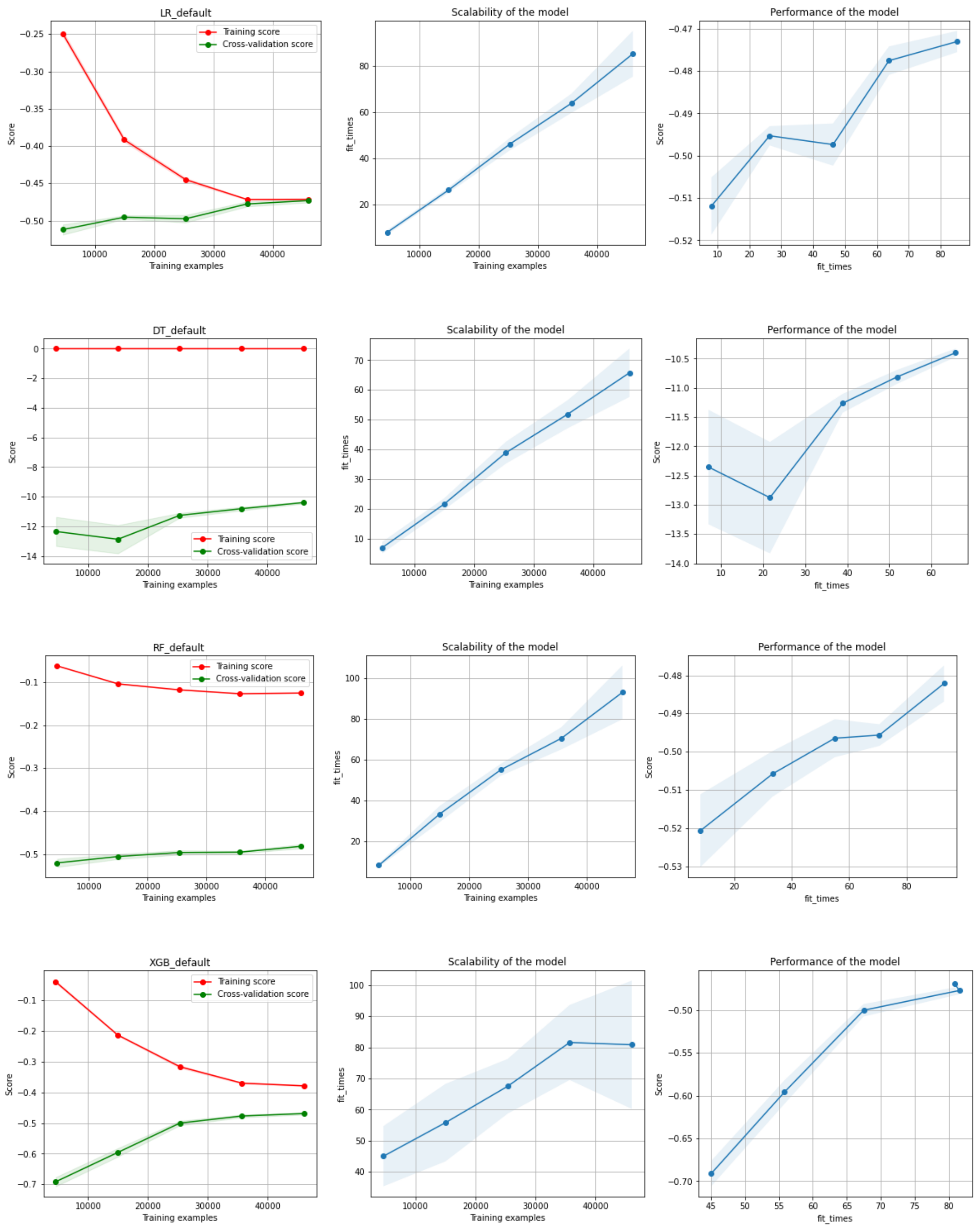


Figure A-18 – Learning curves of the 4 default classification models. Top: Logistic Regression, 2nd from top: Decision Tree, 3rd from top: Random Forest, last: XGBClassifier

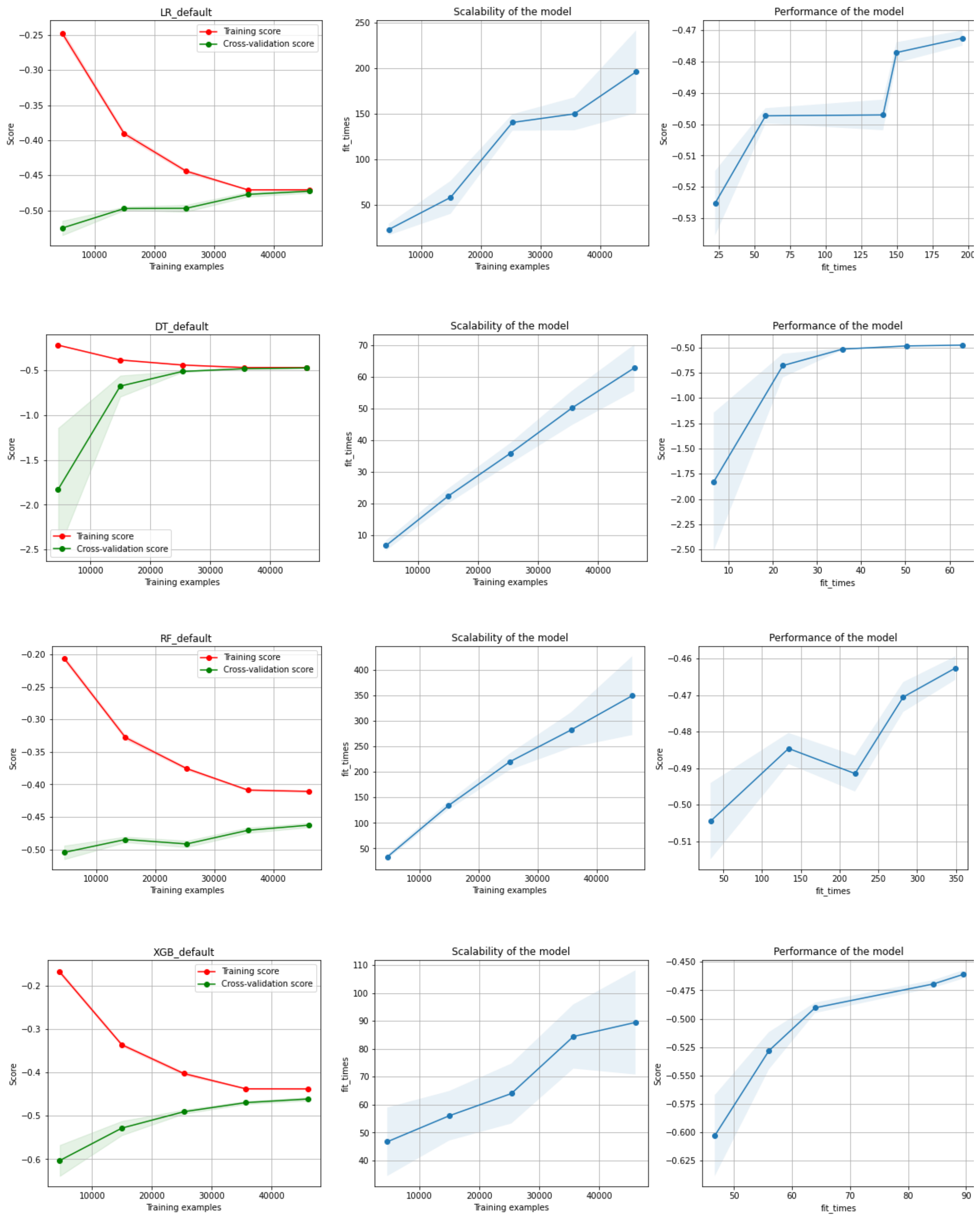


Figure A-19 – Learning curves of the 4 tuned classification models. Top: Logistic Regression, 2nd from top: Decision Tree, 3rd from top: Random Forest, last: XGBClassifier

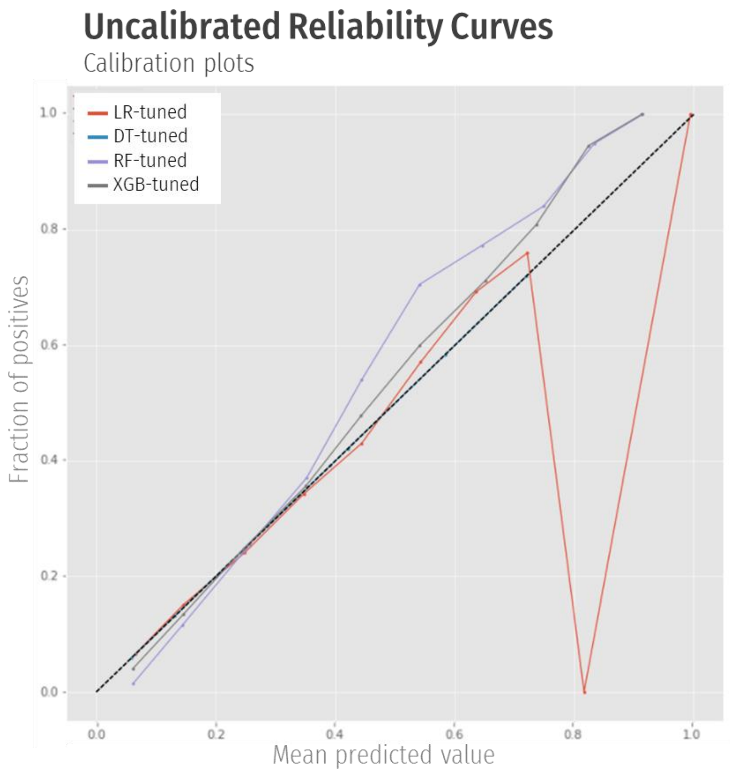


Figure A-20 – Reliability curves of uncalibrated parameter-tuned models.

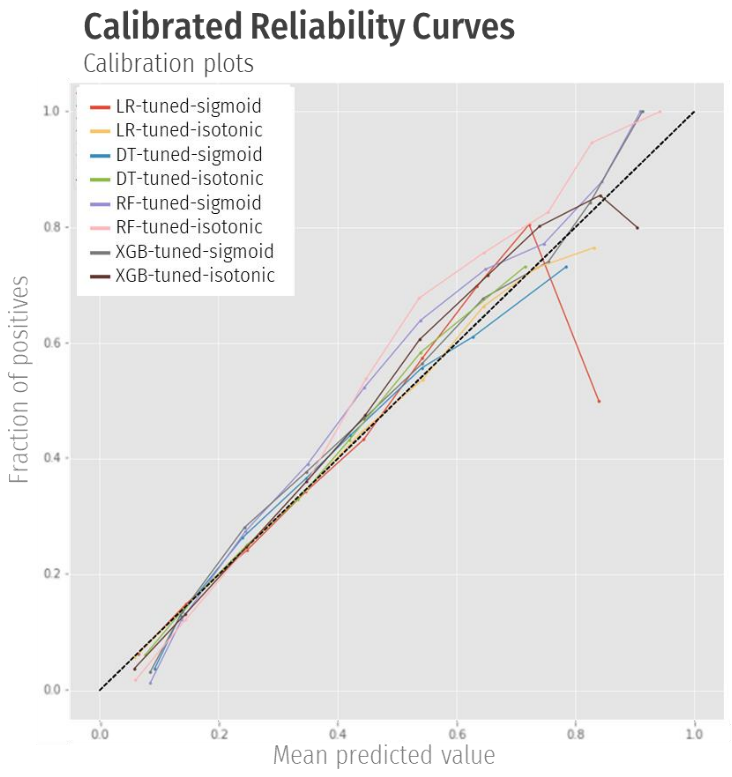
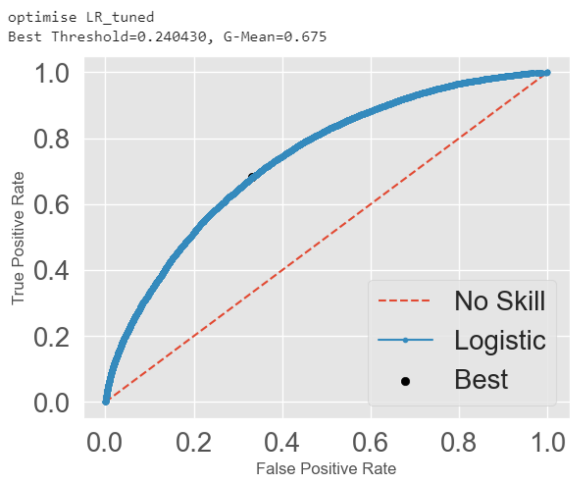


Figure A-21 – Reliability curves of calibrated parameter-tuned models.



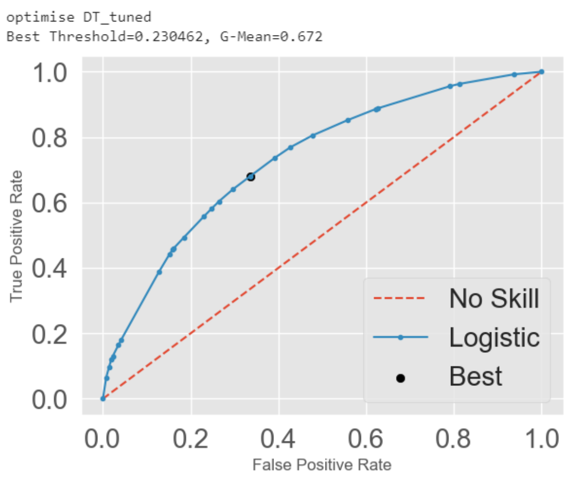
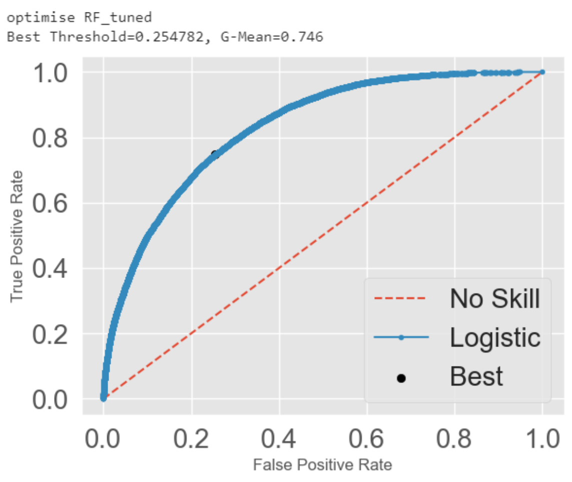


Figure A-22 – Optimisation of probability threshold for LR – Tuned and DT – Tuned models.



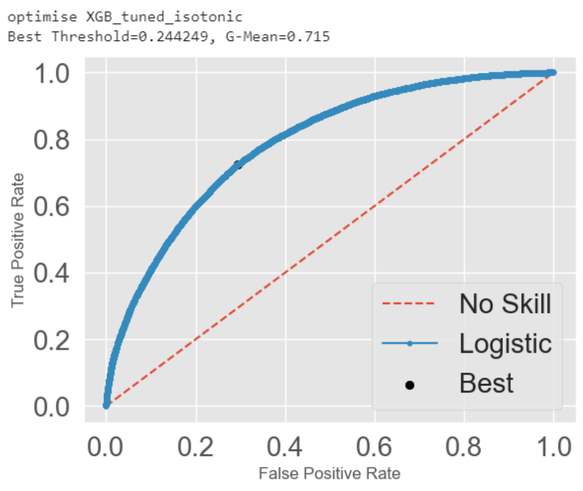


Figure A-23 – Optimisation of probability threshold for RF – Tuned and XGB – Tuned Isotonic models.

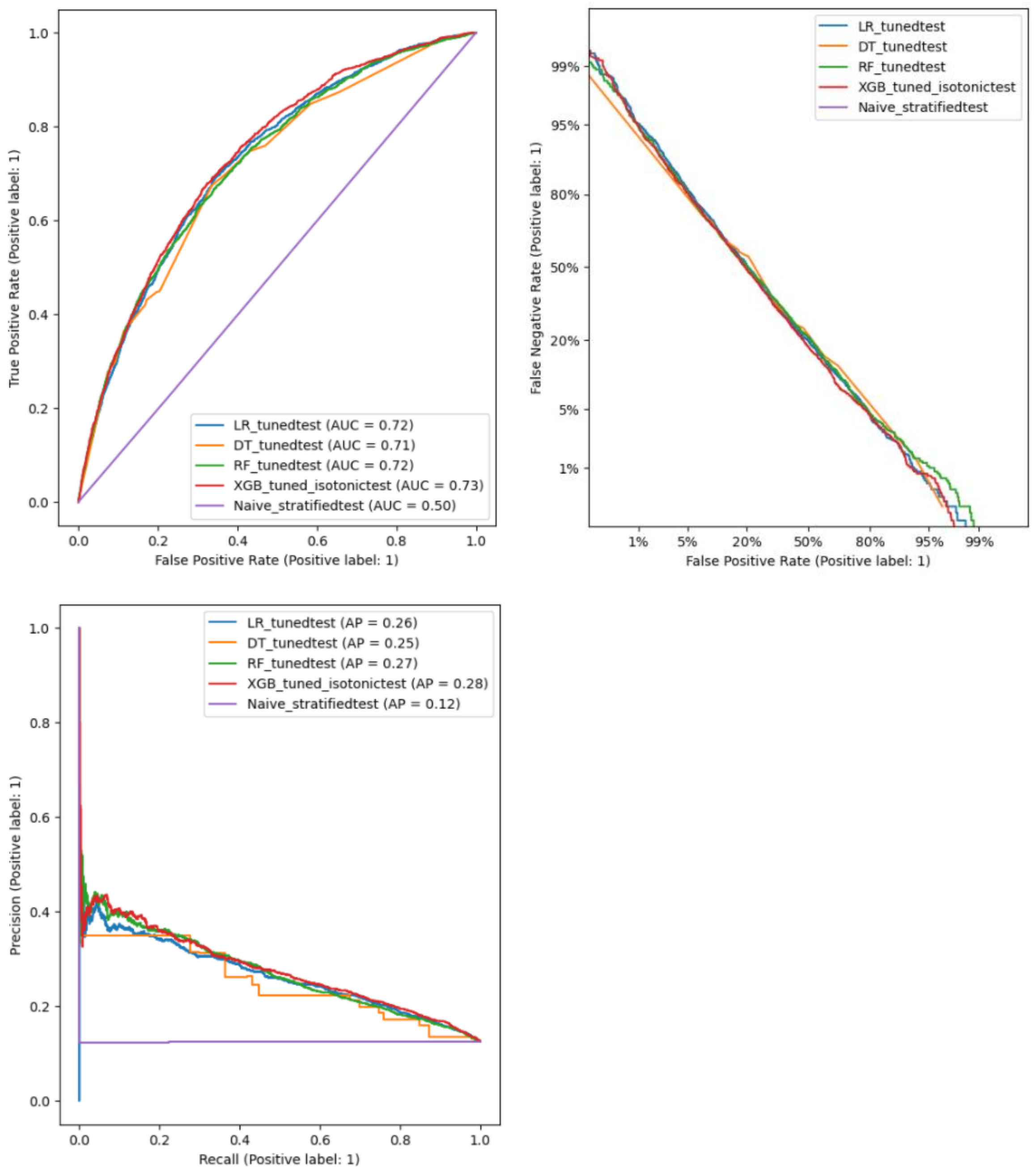


Figure A-24 – ROC (top left), Detection Error Tradeoff (top right) and Precision-Recall curves (bottom) of the final evaluation models

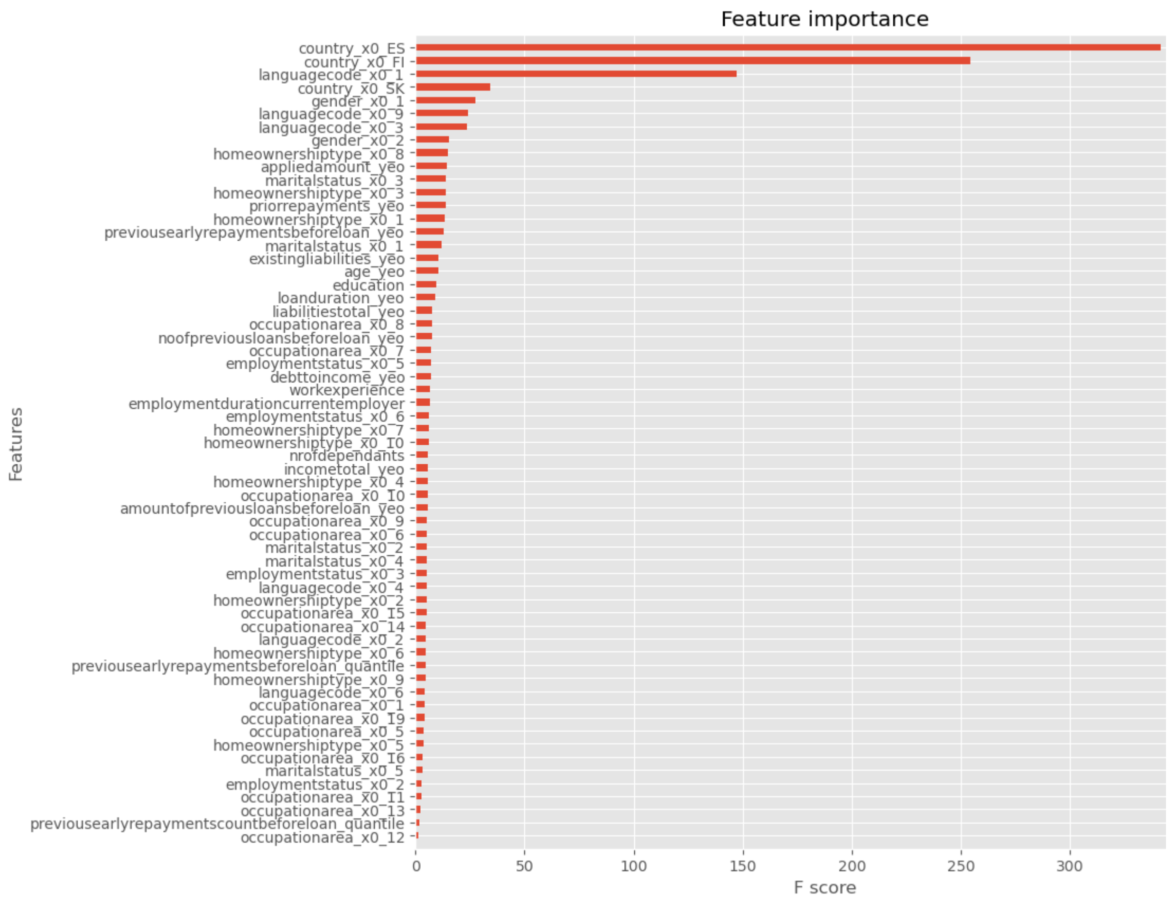


Figure A-25 – Feature importance by gain according to XGB-Tuned Isotonic model.

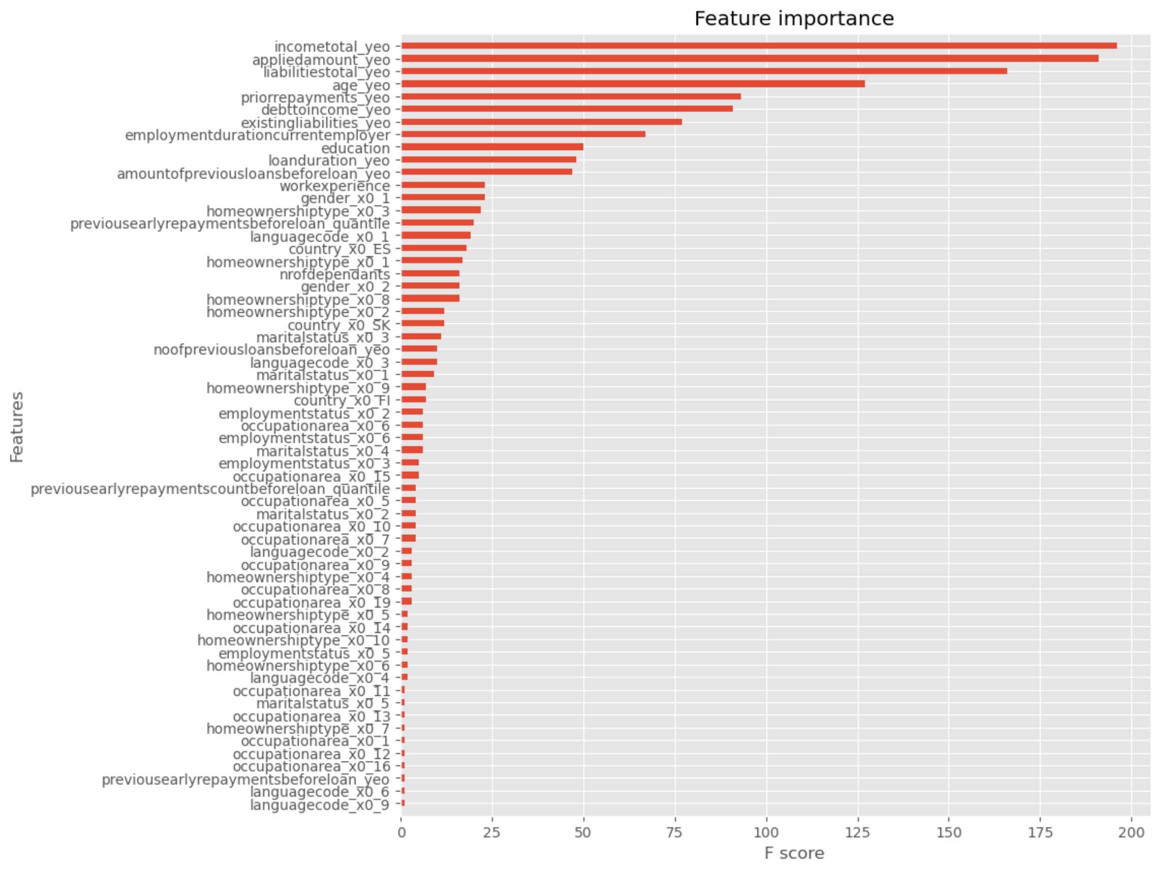


Figure A-26 – Feature importance by weight according to XGB-Tuned Isotonic model.

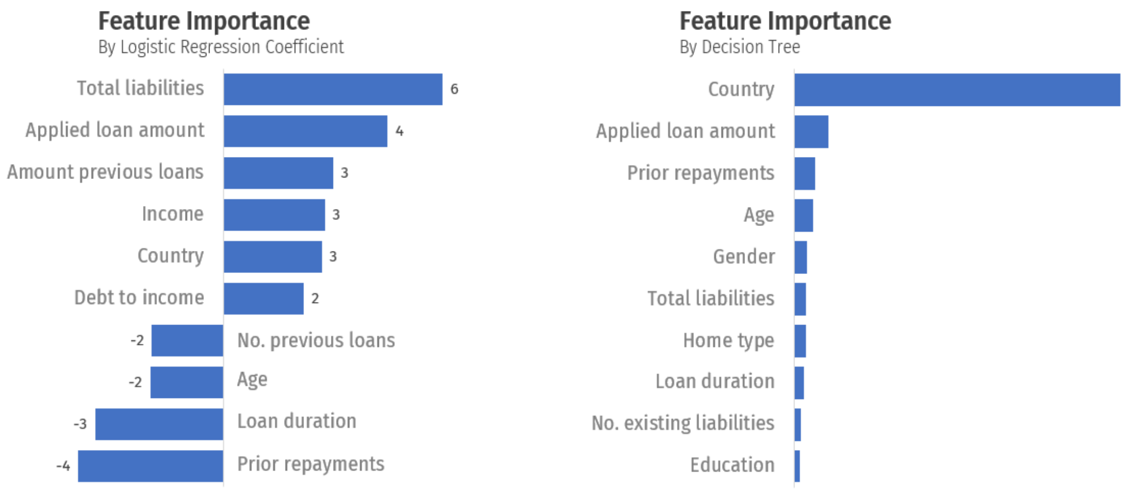


Figure A-27 – Top feature importance by LR-Tuned and DT- Tuned models.

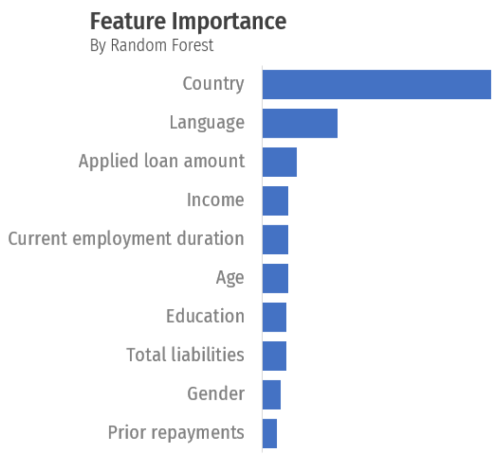


Figure A-28 – Top feature importance by RF-Tuned model.