MTH786P Final Project

Viktor Penchev - 230966576

January 2024

1 Introduction

For my final project, I decided to work on the Credit Card Approval dataset. The dataset contains around 1500 entries of credit card applications and their outcomes. The goal of the project is to implement a binary classifier to predict the outcome of a given application.

2 Analysis of the dataset

2.1 Preprocessing

The data requires some preprocessing before being analysed and used for training models. Here is what I did feature by feature:

1 *Type_Occupation* - this feature had a lot of missing values (488 in total). Because of that, I decided to replace the missing values with value *Unknown* as that can be useful for us to extract more information.

2 *GENDER* - this feature, I treated similarly and replaced the missing values with *Unknown*.

3 *Birthday_Count* - I converted the values to mean how old the person is. Additionally, I replaced the missing values with the mean value as there were only 22 of them

4 *Employed_days* - this feature's values should also be converted to years, however, I forgot to do that. I think that shouldn't affect the model performance in any way and the presence of an auxiliary value to signify unemployment should also not be an issue for the random forest algorithm.

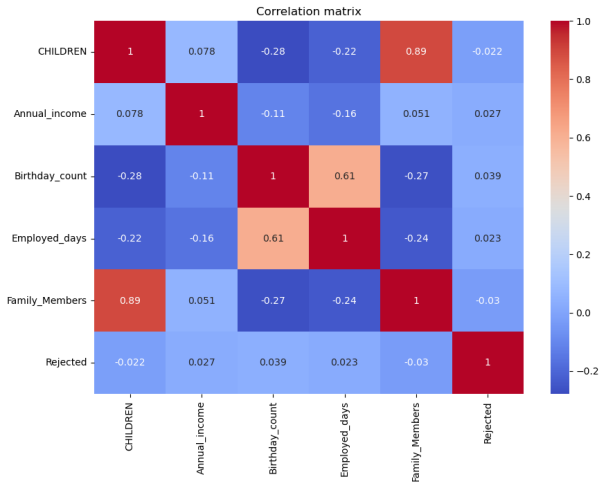
I dropped the rest of the missing values and also the *Ind_ID* column.

2.2 Exploring the variables

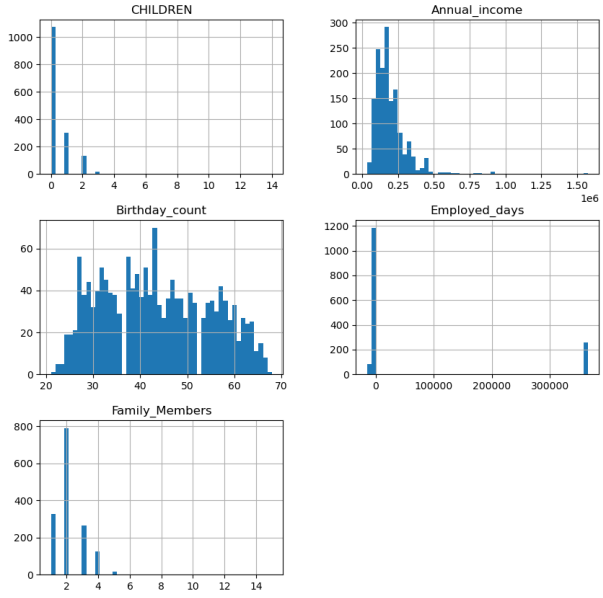
An important part of any data science project is observing the data and exploring any interesting relations even before any type of modelling.

2.2.1 Numerical features

First, I decided to look into the numerical features of the data.

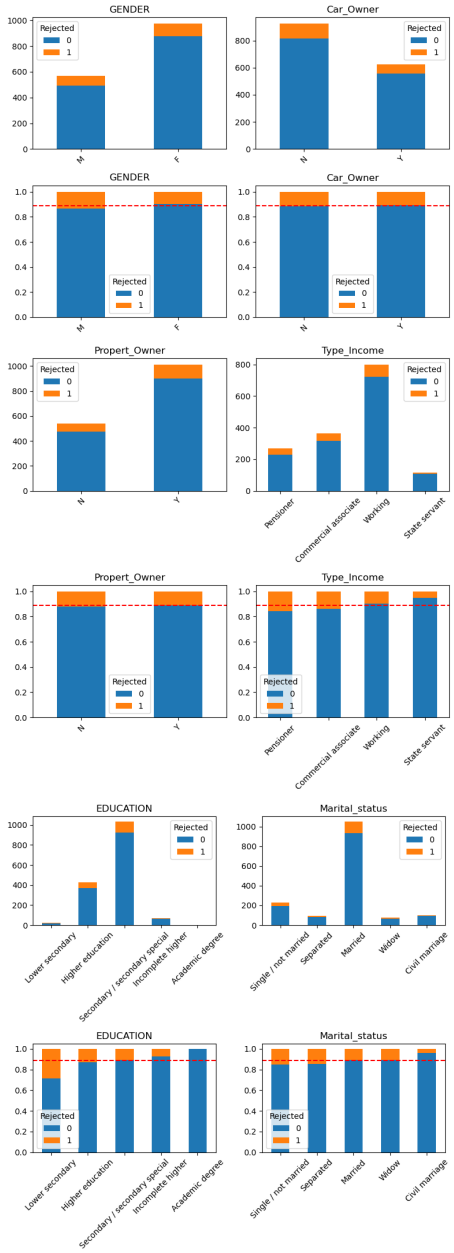


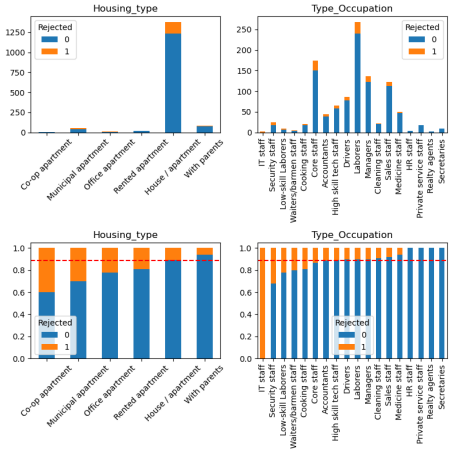
Here I have presented the correlation matrix of the numeric features. Of most importance are the correlations of *Rejected*, however, they appear to be very small and insignificant. This suggests that each piece of information regarding a particular credit card application plays a role in the application outcome and there aren't obvious features which affect the outcome greatly.



Next, I decided to plot the distributions of the numeric features. Here,

again nothing particularly stands out, except the feature *Employed_days* which I forgot to preprocess.





Next, I have plotted the categorical variables. For each feature, I have made two plots - one showing the number of observations of each value, and another showing the proportion of rejects for each value. It is interesting to note that *Car_Owner* and *Property_Owner* seem to not have much effect on *Rejected*. The other values behave as expected, for example, better education is positively correlated with a successful application. Certain charts still seem surprising, for example, all *IT staff* have been rejected, however, it must be noted that that is often the case because of small small sample numbers.

2.3 Output class imbalance

When observing the data, a big issue becomes obvious instantly. That is namely the fact that the target variable is extremely unbalanced - the proportion between *Accepted* and *Rejected* is around *9:1*. This is undesirable as if not treated carefully, it can lead to poor predictive performance of the minority class. This is particularly important in our case because it can lead to many credit card applications being accepted when in fact they should be rejected.

The presence of an underrepresented class leads to further issues in judging the accuracy of the model. In our case, if the model learns to predict *Accept* in each case it will achieve an accuracy of around *90%* which sounds good on paper but will lead to catastrophic outcomes for the bank offering access to credit through credit cards.

In the next section, I will talk about how I handled this issue. It was by far the most challenging part of the problem and even though I achieved adequate results in the end, there surely remain further optimisations to be made.

3 Methods

3.1 Model

For the model, I chose to use a random forest. This decision was made because I wished to learn more about the algorithm and because I expect it to perform well on this kind of tabular data.

For training the random forest, I also created a bootstrap sampling function, which returns the X_{boot} and y_{boot} used for training a tree, as well as the X_{oob} and y_{oob} used for testing.

I used Ethem Alpaydin's *Introduction to Machine Learning* textbook [2] as a source to learn more about the algorithm and its implementation.

3.1.1 A single tree

A random forest is an ensemble of decision trees. A decision tree is in essence a collection of if-then rules which split the sample space into decision regions associated with a given output classification. When presented with a new data point, the if-then rules are applied to it and the algorithm returns the output of the decision area the point is in. The algorithm takes the training the training data and creates an initial rule which is called the **root node**. Each rule is created such that it maximises *information gain* and minimises *entropy*. Entropy is defined as such:

$$H(X) = -\sum_j p_j log(p_j)$$

where X is the subset of data and p_j denotes the probability that a value from X belongs to class j.

Entropy is a measure of uncertainty which takes values between 0 and 1, with 0 being the most certain (all values belong to one class) and 1 being the most certain (all classes are equally likely). In my code, I implemented other impurity measures such as the *Gini index*, and the *Misclassification error*. For my model, I set *entropy* as the default and didn't experiment with changing it.

When presented with the training data, the **root node** is created such that the two splits in the data created have the least amount of entropy. Another way to phrase it is that each **node** aims to maximise information gain, which is defined so:

$$IG(D) = H(D_{parent}) - H(D_{left})\frac{N_{left}}{N_{parent}} - H(D_{right})\frac{N_{right}}{N_{parent}}$$

It is important to note that we multiply the entropies of the left and right split with weights equal to the subsample proportions.

Each **node** after the **root node** is created in the same way until there aren't any more splits to be made (each node has reached an entropy of 0) or some other stopping criterion is reached.

That being said, a full-depth tree should achieve 100% accuracy on the training set (which means that it will likely overfit). In our case, however,

I noticed that wasn't quite the case and it turned out that there are points with the same values for all features except the target feature, *Rejected*. This observation wasn't included in the code provided, but I decided to mention it here instead.

3.1.2 A random forest

A random forest is a collection of decision trees. Each tree is trained in the same way, however, the training data varies between trees. This way, the trees can together learn a more complete representation of the data. Prediction works by having each tree make a prediction and then taking the majority outcome.

The variance in training data and therefore between the individual trees comes from the *bootstrap sampling* technique used to split the data into training and testing subsets. It works on the principle of drawing random samples from the data *with replacement* until a number of points equal to the length of the original dataset is drawn. These samples which were drawn constitute the training data and all the samples not drawn are left as testing data. When training a random forest, bootstrapped samples are drawn for each tree from the forest and this way each tree gets to learn to classify different subsets of the data.

In my implementation of a random forest, I included the following parameters:

- *n_trees* - the number of trees which constitute the random forest (100 by default).
- *max_features* - an integer representing the number of random features between which the optimal split can be considered at a given node.
- *max_depth* - the maximum allowed depth of each tree.
- *min_samples_split* - the minimum amount of samples which allows a new split to be made.
- *decision_threshold* - threshold for percentage of *Rejected* predictions above which we predict *Rejected* (0.5 by default but I experimented with lower values to account for the data imbalance).

3.2 K-fold cross-validation

To validate the results of the models, I decided to use *K-fold cross-validation*. It works by splitting the available data into n splits and training the model n times. Each time the model is trained, one of the splits is left as a test set while the rest are used for training.

When validating a random forest using k-fold, the bootstrapping function is used on the training splits and the test split functions as a separate validation to the bootstrapping test.

After a model is trained on one of the splits, we get the accuracy achieved. The final accuracy is calculated by taking the average over all the splits. In my experiments I always set the number of splits to be equal to 4. This was to ensure a big enough size of the test splits and an adequate representation of the minority target class.

3.3 Addressing the data imbalance

As mentioned before, a big challenge of this classification task is the imbalance in the target variable classes. One of the remedies I considered was the use *decision_threshold* parameter. When the parameter is lowered, the trees in the forest are biased to predict *Rejected* in the sample space partitions where training samples of both classes are present. This makes it so that the trees prioritise True Positives (predicting *Rejected* when the actuality is *Rejected*) and False Positives (predicting *Rejected* when the actuality is *Accepted*) over True Negatives (predicting *Accepted* when the actuality is *Accepted*) and False Negatives (predicting *Accepted* when the actuality is *Rejected*).

I implemented two further measures to improve results with the data imbalance in mind.

3.3.1 F1 Score

In order to evaluate the model performance more accurately, I implemented the F_1Score measure of predictive performance. The F_1Score is defined as so:

$$F_1 = \frac{2TP}{2TP + FP + FN}$$

The measure is the harmonic mean of *precision* and *recall* where:

$$precision = \frac{TP}{PredictedPositives}$$

$$recall = \frac{TP}{ActualPositives}$$

The measure is more useful in our case as normal accuracy can be unreliable due to the class imbalance.

When evaluating the models I trained, I aimed at achieving the highest possible F_1 while ensuring the *accuracy* doesn't suffer.

3.3.2 Synthetic Minority Oversampling Technique (SMOTE)

The last method I implemented to fight the data imbalance is the *Synthetic Minority Oversampling Technique (SMOTE)*[1]. The *SMOTE* algorithm, as described in the paper, takes in three inputs: T - samples of minority class, N - SMOTE %, and k - number of nearest neighbours.

The algorithm goes through each point in T, calculates the k nearest neighbors from the minority class, and creates $int(N/100)$ new samples by making

new points between the current sample and a random selection of the k nearest neighbours.

These points can then be used as part of the training set to increase the proportion of the underrepresented class. I had to make sure that in both the *bootstrap sampling* and the *k-fold cross-validation*, the included generated sample would only be added to the training sets and never to the test sets as they are, of course, fabricated and will make the results unreliable.

4 Results

I tried various combinations of parameter values, with and without the use of artificial data. The parameter *n_trees* was always left at 100. Training the forests became very time-consuming due to the *k-fold cross validation* and I didn't have the chance to fine-tune the parameters to perfection. Nevertheless, here are my efforts and results.

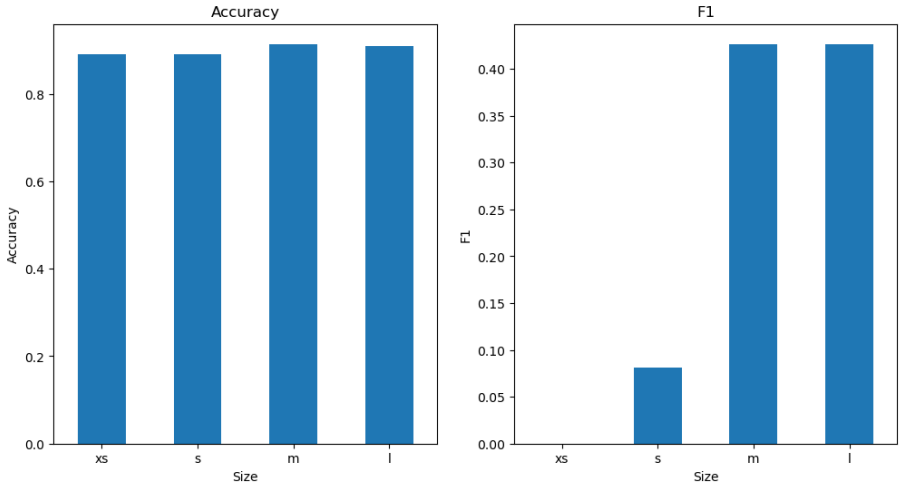
4.1 Experiment 1: Trying out different tree sizes

The first thing I tried was to train random forests with trees of different sizes. I trained the following 4 models:

Name	*max_features*	*max_depth*	*min_samples_split*
RandomForest_XS	3	5	10
RandomForest_S	5	10	5
RandomForest_M	7	15	2
RandomForest_L	10	20	2

Before moving on to the results, I want to briefly mention a few things:

- The parameter *min_samples_split* can take a value of 2 at the least as that means that when 2 samples of different classes have reached a node, the node will be split if it is possible.
- When training a single full-depth tree on all the data, it was of depth 22. That means that the largest tree is more or less uncapped in terms of size.



It is clear that forests with larger trees perform better. These results suggest that smaller trees can't learn enough nuance to distinguish the rare *Rejected* cases. The best model seems to be the M and a larger network appears to not lead to any increase in performance.

The results of the best model (M) are: $Accuracy = 0.91$ and $F_1 = 0.43$.

4.2 Experiment 2: The lowering of the *decision_threshold* parameter

Next, I wished to explore the use of the *decision_threshold*. Unfortunately, at the very last moment I realised I had a bug and was unable to run the whole experiment again. Even though the code is correct currently, I simply didn't have time to let the models train and see the results.

For the experiment I wished to train 4 random forests, all of the same size: $max\_features = 5$, $max\_depth = 10$, $min\_sample\_split = 2$, but with different values of *decision_threshold* (between 0.5 and 0.1).

4.3 Experiment 3: Best performing model

To find the best model, I wanted to test 6 different random forests. As each one took around 40 minutes to validate using *k-fold cross-validation*, I was again unable to conduct the whole experiment. My plan was to train 9 models, each of the same size as model M. Each three forests were planned to have the same number of artificial samples introduced (those being 0, $2 * 128 = 256$, and $4 * 128 = 512$) but different values for *decision_threshold*. In the end, I managed to train only three of the models. Here are their results:

SMOTE N	*decision_threshold*	*Accuracy*	$F_1 Score$
2	0.5	0.93	0.64
2	0.2	0.72	0.41
4	0.5	0.90	0.64

It is clear that the two models with default *decision_thresholds* did better than the one with $decision\_threshold = 0.2$. I think the use of both *SMOTE* for generating artificial data and a low *decision_threshold* lead to the model being too biased towards predicting *Rejected*. Because of this, we can see that both the accuracy and the $F_1 Score$ of the 2nd model are low.

Another interesting observation is that the introduction of artifical samples has enabled us to improve upon the previously best model (the *M-sized* model). This has left us with the two best-performing models: *M-sized models with 2*128 or 4*128 artificial samples introduced and default decision_threshold.*

5 Conclusion

This dataset posed a challenging classification task which I had a great deal of excitement working on. The main problem I faced was dealing with the imbalanced target classes. The *SMOTE* algorithm I used for creating artificial data to balance out the ratios between the two target classes showed great improvement in the $F_1 Scores$ of the predictions.

Unfortunately, in the end I didn't have enough time to thoroughly compare results and conclusively find a best-performing model. Nevertheless, I got key insights from the few experiments I managed to conduct fully:

- For this dataset, a forest of deep trees showed better results than those with *prepruned* ones.
- Introduction of artificial samples increased the $F_1 Score$ without penalising accuracy.
- Lowering the *decision_threshold* when artificial samples were present seems to make the model too biased in predicting *Rejected*. This means that the *recall* of the model is high but the *precision* is low and this is reflected in a low $F_1 Score$.

References

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

[2] Alpaydin, E. (2014). Introduction to machine learning. Cambridge, Massachusetts: The MIT Press.