Give Me Some Credit: Nutritional Labels for Automated Decision Systems

Michelle Espinoza (me1771)

Lin Shue (kls678)

Table of Contents

1.1 Background

Banks play a crucial role in market economies. They decide who can get finance and on what terms and can make or break investment decisions. For markets and society to function, individuals and companies need access to credit. Credit scoring algorithms, which make a guess at the probability of default, are the method banks use to determine whether or not a loan should be granted. Give Me Some Credit is a program that aims to improve on the state of the art in credit scoring by predicting the probability that somebody will experience financial distress in the next two years. A total of 10 distinct variables were given in the dataset to be used as predictors, including age, monthly income, debt ratio, etc. 250000 borrowers' historical data were provided as two sets of data, one as training dataset with 150000 of the borrowers with the desired prediction answer and 100000 of the borrowers as the test dataset without the desired prediction.

In the training dataset:

- 37500 were aged below 41 (25%)
- 75000 were aged above 52 (50%)
- 25% have a monthly income that is lower than 3400
- 50% have a monthly income between 3401 and 8249
- 58% do not have any dependent
- 41474 have more than 1 real estate loan (28%)

The purpose of this ADS is to be able to accurately determine who will experience financial distress within the next two years, which will then help banks determine whether a loan should be granted and therefore further improve upon the system of credit scoring. The stated goals of this ADS is to help borrowers make the best financial decisions.

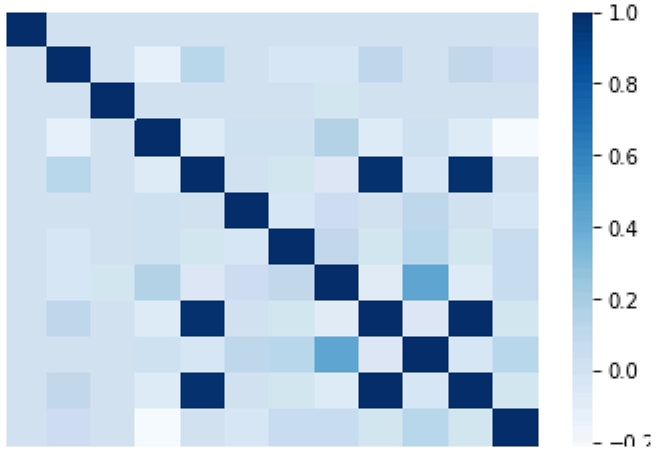
1.2 Input and Output

Give Me Some Credit Data

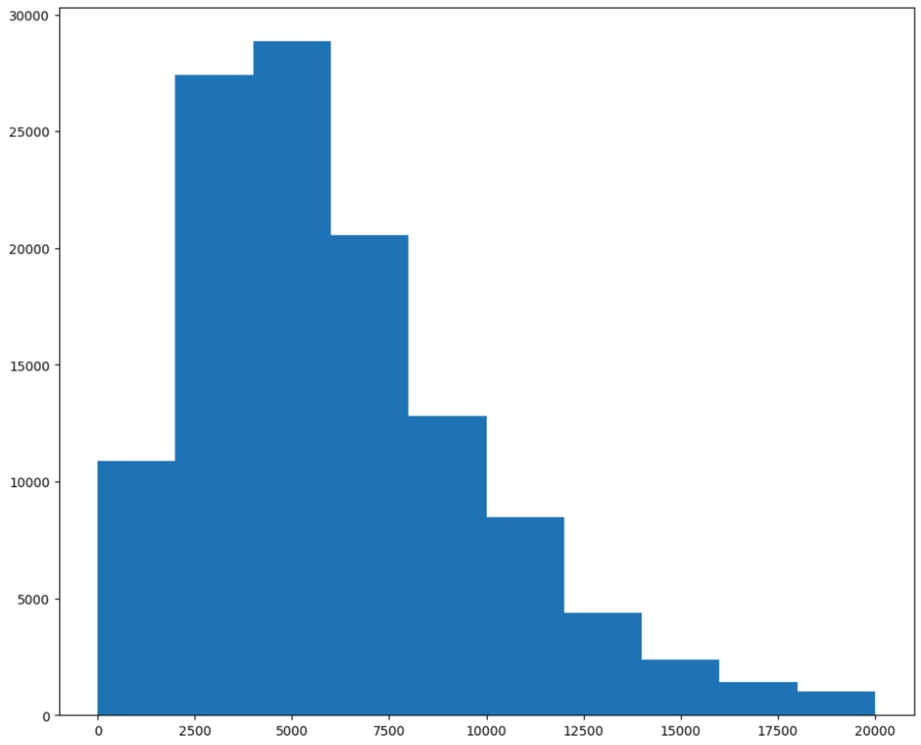
- The training dataset shape is (150000, 12) and the test dataset shape is (101503, 12)
- The data used by this ADS is historical data collected from 250,000 borrowers. This data consists of integers and floats (percentage), with instance monthly income as integer and debt ratio as float.

Features	**Data Type**
SeriousDlqin2yrs	Boolean
RevolvingUtilizationOfUnsecuredLines	Float
Age	Integer
NumberOfTime30-59DaysPastDueNotWorse	Integer
DebtRatio	Float
MonthlyIncome	Integer
NumberOfOpenCreditLinesAndLoans	Integer
NumberOfTimes90DaysLate	Integer
NumberRealEstateLoansOrLines	Integer
NumberOfTime60-89DaysPastDueNotWorse	Integer
NumberOfDependents	Integer

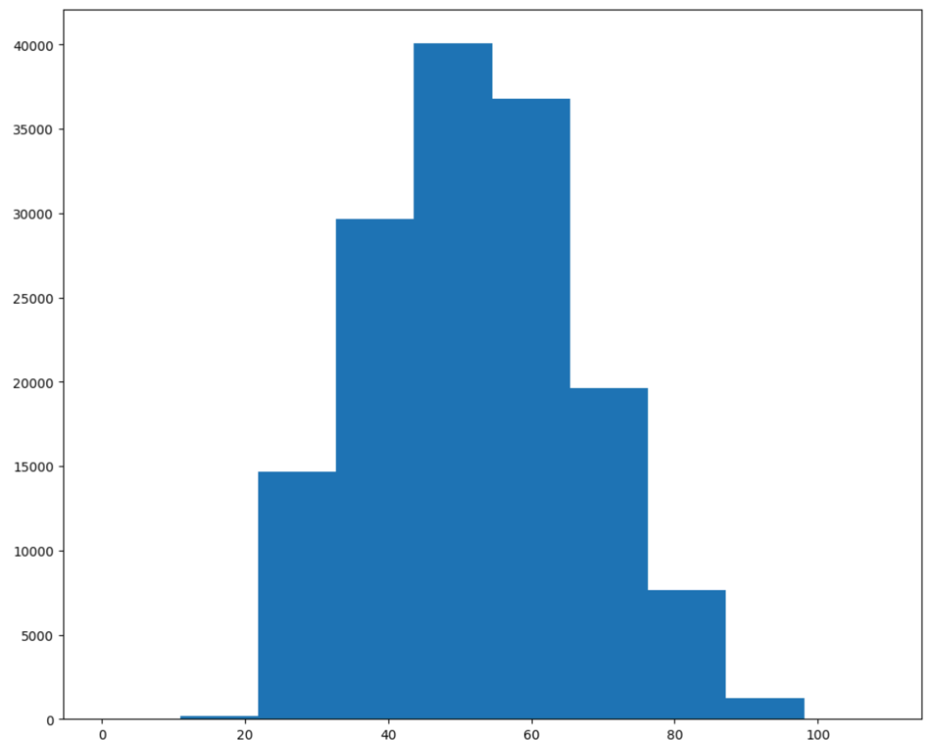
- There are a total of 11 input variables, only 2 variables contain missing values, which is monthly income and number of dependents. The correlations between features and a selection of variables' distribution from the ADS are shown below:



Correlation between each variable in training dataset



Distribution of MonthlyIncome



Age Distribution

- All models used in the ADS are evaluated based on accuracy, furthermore, the ADS does not perform inference, that is, it does not use any of the trained models for classification on unseen data.

1.3 Implementation and Validation

For this ADS, there were various steps taken to clean and pre-process the data. After first loading the data, the data's dtypes were then converted in order to ensure that there are no errors due to using the inappropriate data types. Then, by iterating through the columns of the dataframe and modifying the datasets, the memory usage was decreased in order to simplify the dataset.

After simplifying the data, in order to clean the dataset, an iterative imputer was created in order to correct skewness (therefore reducing bias) and predict values that could replace the null values in monthly income. To do so, datasets for the training and testing datasets were created by taking the rows of data where the values for monthly income were null and then dropping the columns for monthly income. The remaining data was then used to predict the values that would replace the null value for monthly income. Afterwards, the null values for monthly income in the original training and testing datasets were replaced with the predicted values, ensuring that the original datasets were now cleaner. The same process was repeated for the null values for the number of dependents.

Once the data was cleaned, the numeric values of the data were discretized using binning, allowing the data to consist of categorical data. After discretizing the data, the data is put through the pipeline (allowing us to automate the machine learning process), where the missing values were replaced with the median. After discretizing and pipelining the data, the data was then fitted to the training set, completing the data pre-processing.

We then select the y-values (the values we want to predict), titling them as **SeriousDlqin2yrs** and the x-values(values used to make the prediction), which consists of the numerical and categorical data of the test dataframe.

We then separate the columns of X by their categorical dtypes and numerical dtypes and create the features engineer class, which will be used to cluster similar data and then take their k-means in order to extract features from the data that are suitable for machine learning. The following demonstrates the category values used for each feature, with their numerical value equivalent indicated in parentheses:

Categories for MonthlyIncome: 0.0 (< 0.5), 1.0 (< 1000), 2.0 (<= 1500), 3.0 (< 2000), 4.0 (< 2300), 5.0 (<= 2500), 6.0 (< 3000), 7.0 (< 3200), 8.0 (< 3400), 9.0 (< 3700), 10.0 (< 3900), 11.0 (<= 4000), 12.0 (< 4400), 13.0 (< 4600), 14.0 (< 4900), 15.0 (<= 5000), 16.0 (< 5400), 17.0 (< 5700), 18.0 (<5900), 19.0 (<= 6200), 20.0 (< 6600), 21.0 (< 7000), 22.0 (< 7300), 23.0 (< 7800), 24.0 (< 8300), 25.0 (<= 8700), 26.0 (< 9400), 27.0 (< 10000), 28.0 (<= 10700), 29.0 (< 12000), 30.0 (< 14000), 31.0 (<= 16000), 32.0 (< 3009000.0)

Categories for DebtRatio: 0.0 (<= 0.00400), 1.1 (<=0.016000), 2.0 (<=0.03), 3.0 (<=0.052), 4.0 (<=0.085), 5.0 (<=0.12), 6.0 (<=0.14), 7.0 (<=0.15), 8.0 (<=0.17), 9.0 (<=0.19), 10.0 (<= 0.21), 11.0 (<= 0.23), 12.0(<= 0.25), 13.0 (<= 0.27), 14.0 (<= 0.28), 15.0 (<= 0.30), 16.0 (<= 0.32), 17.0 (<= 0.34), 18.0 (<= 0.37), 19.0 (<= 0.39), 20.0 (<= 0.41), 21.0 (<= 0.44), 22.0 (<= 0.46), 23.0 (<= 0.49), 24.0 (<= 0.53), 25.0 (<= 0.58), 26.0 (<= 0.64), 27.0 (<= 0.73), 28.0 (<= 0.85), 29.0 (<= 1.1), 30.0 (<= 2.8), 31.0 (< 40), 32.0 (< 215), 33.0 (< 700), 34.0 (< 1300), 35.0 (< 1750), 36.0 (< 2220), 37.0 (< 3300), 38.0 (< 34000)

Categories for age: 0.0 (<= 40), 1.0 (<= 51), 2.0 (<= 62), 3.0 (<= 100)

Categories for NumberOfDependents: 0.0 (<= 0.5), 1.0 (>= 1.0)

Categories for NumberRealEstateLoansOrLines: 0.0 (< 1.0), 1.0 (< 2.0), 2.0 (<= 8.0)

Categories for RevolvingUtilizationOfUnsecuredLines: 0.0 (< 0.0001), 1.0 (< 0.003), 2.0 (< 0.006), 3.0 (< 0.01), 4.0 (< 0.015), 5.0 (< 0.02), 6.0 (< 0.025), 7.0 (< 0.03), 8.0 (<

0.037), 9.0 (< 0.042), 10.0 (< 0.051), 11.0 (0.06), 12.0 (< 0.071), 13.0 (< 0.082), 14.0 (< 0.097), 15.0 (< 0.12), 16.0 (< 0.14), 17.0 (< 0.16), 18.0 (< 0.18), 19.0 (< 0.21), 20.0 (< 0.24), 21.0 (< 0.28), 22.0 (< 0.31), 23.0 (< 0.35), 24.0 (< 0.40), 25.0 (< 0.45), 26.0 (< 0.50), 27.0 (< 0.56), 28.0 (< 0.62), 29.0 (< 0.70), 30.0 (< 0.77), 31.0 (< 0.86), 32.0 (< 0.92), 33.0 (< 0.98), 34.0 (< 1.0), 35.0 (< 1.1)

Categories for NumberOfTime60-89DaysPastDueNotWorse: 0.0 (all)

Categories for NumberOfTimes90DaysLate: 0.0 (all)

Categories for NumberOfOpenCreditLinesAndLoans: 0.0 (<= 4.0), 1.0 (<= 7.0), 2.0 (<= 10) 3.0 (<= 58.0)

Categories for NumberOfTime30-59DaysPastDueNotWorse: 0.0 (all)

We now select the final pipelines consisting of a categorical transformer pipeline (which imputes missing values with the mode and encodes the variables using weight of evidence), a numeric transformer pipeline (which imputes missing values with the median, scales the data to make it more Gaussian-like, and uses the feature engineering class used earlier), and a column transformer for the numeric values.

Now we begin creating models using a histogram-based gradient boosting classifier (**HGBC**), which will train faster decision trees. We place this model in a pipeline (making sure to make a copy as well) that will encode the variables using weight of evidence (**WOE**). Another model is created using the **CatBoost Classifier**, which works similar to an **HGBC** but with a higher level of accuracy. This model is also placed in a pipeline that encodes using **WOE**. A third model is created using the **XGBoost Classifier**, which combines the results of several modes in order to make predictions. This model is placed through a similar pipeline as the previous two models. A fourth model is created using LightGBM classifier (**LGBM**), which is a “tree based learning algorithm” that grows the trees vertically, making it more efficient and faster at training. This model is also placed in a similar pipeline as the other three models. Finally, a pipeline that both encodes the variables using **WOE** and models the continuous features conforming to a Gaussian distribution is created (using **GaussianNB**).

After completing these steps, we begin choosing the best pipes by going through the same pipelines and transformers as in previous steps, as well as creating different pipelines with differing percentiles in order to select features according to the 93rd, 98th, 95th, and 90th percentiles of the highest scores. The best pipes selected are the pipelines using the 95th percentile as well as **LGM** classifiers, and the categorical transformer pipelines using the **CatBoost Classifier**. Overall, 5 best pipes were chosen. After choosing the best pipes, we create mlxtend models using the best pipes as classifiers in the **StackingCVCClassifier**, which uses cross-validation to prevent overfitting. We removed the random-state parameter from the **StackingCVCClassifier** since it kept giving an error (though in the original solution code, random-state = 42) when attempting to create the mlxtend models. We then define parameters and use them to create two logistic regression models.

We stack the models by using the **StackingCVCClassifier** with each logistic regression model serving as a **meta_classifier** and then choose a final numerical transformer pipeline and a final column transformer.

We begin the process of oversampling and undersampling in order to ensure the dataset is balanced. We begin undersampling by making four different pipelines using different methods to get estimators, two that utilize the **EditedNearestNeighbours** method (which cleans the dataset by removing samples close to the decision boundary), one utilizing the **OneSidedSelectionMethod**, and one utilizing the **TomekLinks** (which removes noisy and borderline majority class examples). Oversampling is done by using five different methods including **RandomOverSampler**, **SMOTE** (which balances class distribution by randomly increasing minority class examples by replicating them), **ADASYN** (which generates synthetic data), **BorderlineSMOTE** (where Borderline samples will be detected and used to generate new synthetic samples) - which is used twice- and **SVMSMOTE** (which detects sample to use for generating new synthetic samples as proposed).

We then begin cost-sensitive learning by creating two models using the **LGBMClassifier**- one that is balanced, and the other is unbalanced- as well as establishing two different sample weights. To begin pre-processing, we fit X and y according to the preprocessor transformer, and then proceed to transform both the **X** and **X_test**, and then fit resampling **x_pred** and **y** through the random oversampling.

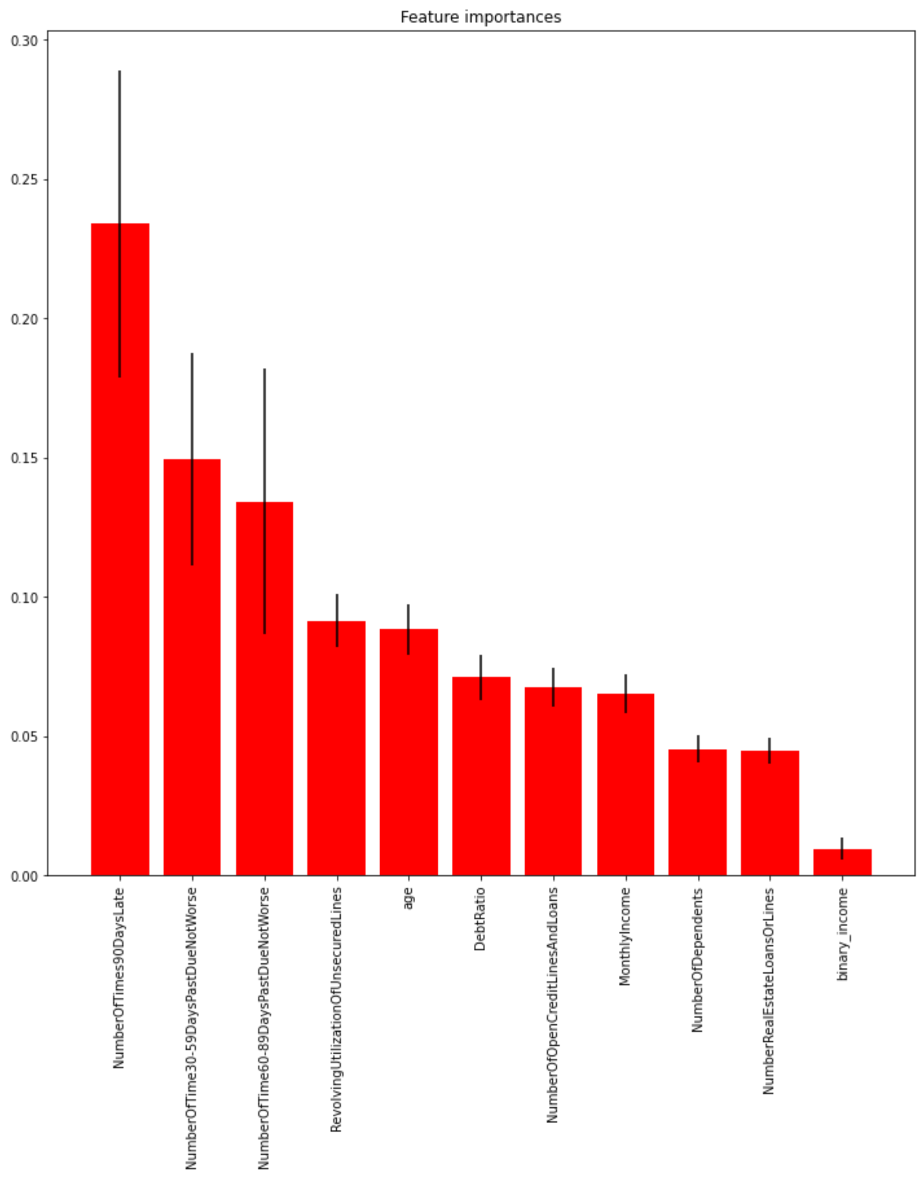
At this point, the code solution started displaying an error due to Colab's memory/runtime limits. As a result, we had to begin analyzing a much less extensive and more simple solution that utilized three predictors: **logistic regression**, **random forest**, and **XGBoost**. Due to time constraints and for the sake of convenience, we will focus on logistic regression.

This solution begins by defining a function called **evalBinaryClassifier**, which is supposed to visualize the performance of a logistic regression binary classifier. However, because the solution only utilizes the **XGBoost** predictor as a parameter of this classifier function, we will not be focusing on its implementation. After defining **evalBinaryClassifier**, we print the shape of the training data, count the values of the training data's dtypes, and print the data's information.

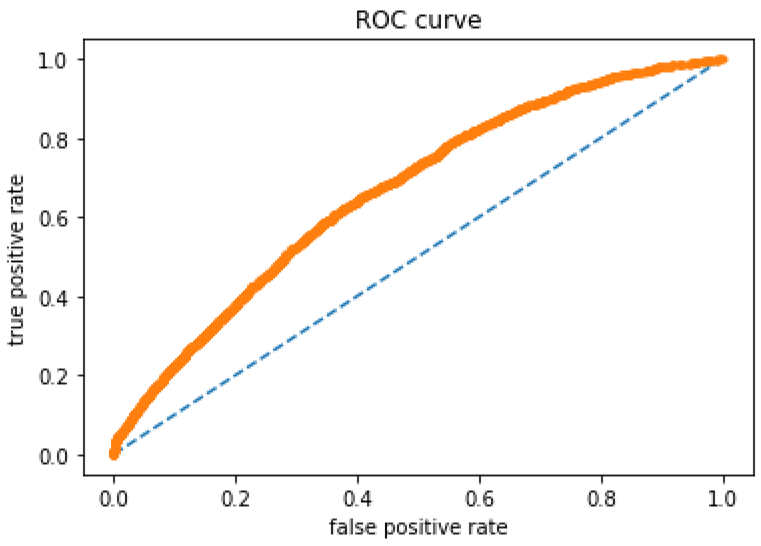
Afterwards, we drop the null values and the first column of the training data, making the shape (**120269, 11)** and create an array that will hold the name of each column (the column names are used as features), the most commonly recurring value, and the number of times that value was counted. We then iterate through the columns of the data in order to count how many times each of the values appears, appending the column name, most frequent column value, and the values' counts to the array we created. Using this array, we create a dataframe displaying the information of the array, sorting the values in order of highest to lowest value counts.

We create x and y values from the training data, creating **train_x** by dropping the **SeriouslyDlqin2yrs** column and creating train_y by obtaining the natural logarithmic values of **SeriouslyDlqin2yrs**.

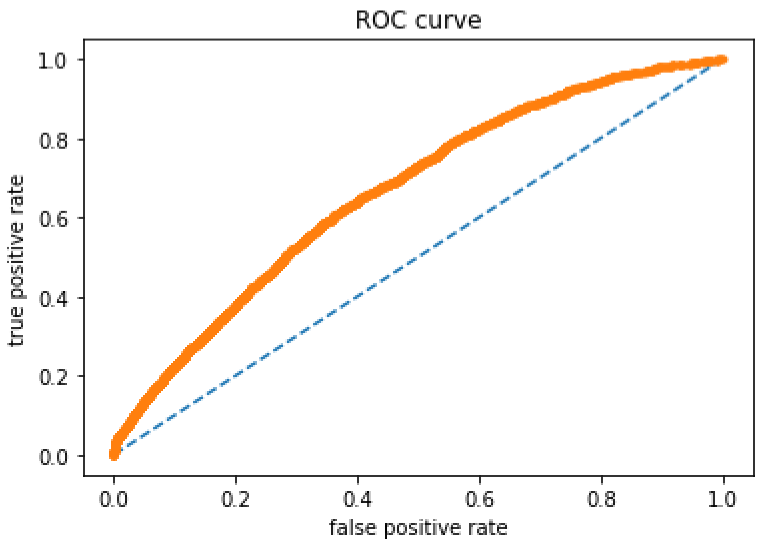
We plot the importance of each feature, which demonstrates that **NumberOfTimes90DaysLate** and other features relating to late payments are most important in predicting delinquency (which is supported by logic).



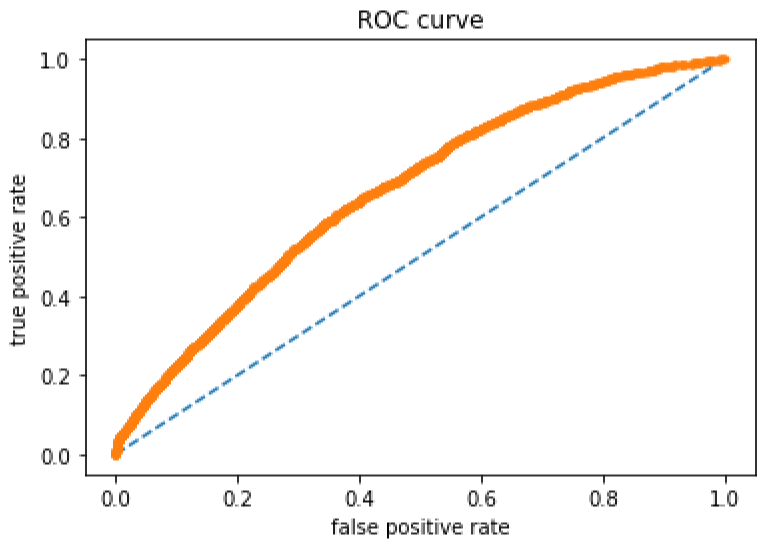
We begin defining a function **plot_roc**, which takes **y_test** as the y values and the probability estimates as parameters in order to plot the **ROC** curve. We also define a function called **logistic**, which essentially takes the training data- dropping the **SeriouslyDqqin2yrs** column- as **x** and **y_train** (transforming the **SeriouslyDlqin2yrs** values of the training data to uint8 dtype) as y and performs **train_test_split** on both x and y. In order to predict and model y, we use the **LogisticRegression()** classifier, fitting x and y to this model and then predict the class probabilities of **X_test** (which we got from performing **train_test_split** on the data) as **probs**, and **y_test**. The function then returns the **ROC AUC** score at 0.6662572612580925, indicating average accuracy. We then plot the **ROC** curve using **probs** and **y_test**:



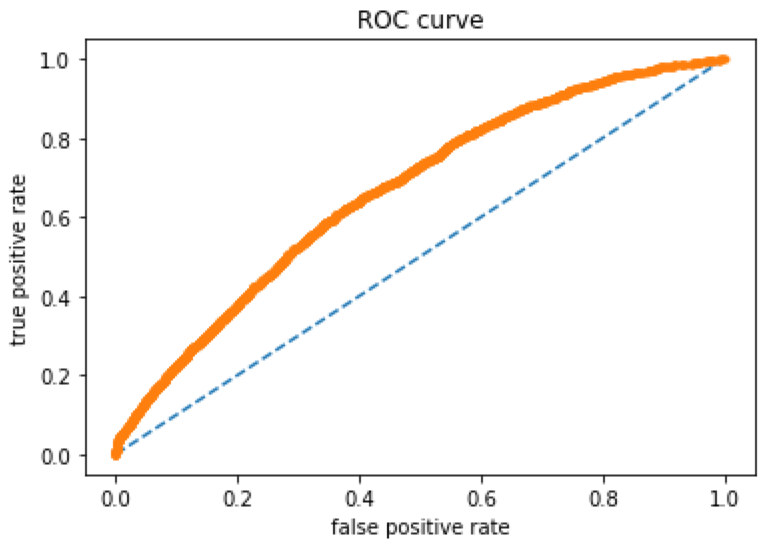
We perform a similar process again, performing **train_test_split** on the same x and y we split previously, fitting x and y to the logistic regression model. We now predict the actual classes using **val_x** (which is like the **X_test** we obtained previously from performing **train_test_split** on the data), rather than the class probabilities- this results in the prediction of y as **pred_y**. We now get the **ROC AUC** score of 0.5041934495861404, indicating below average accuracy, using **val_y** as the actual values of y and **pred_y** as the predicted y and plot the **ROC** curve:



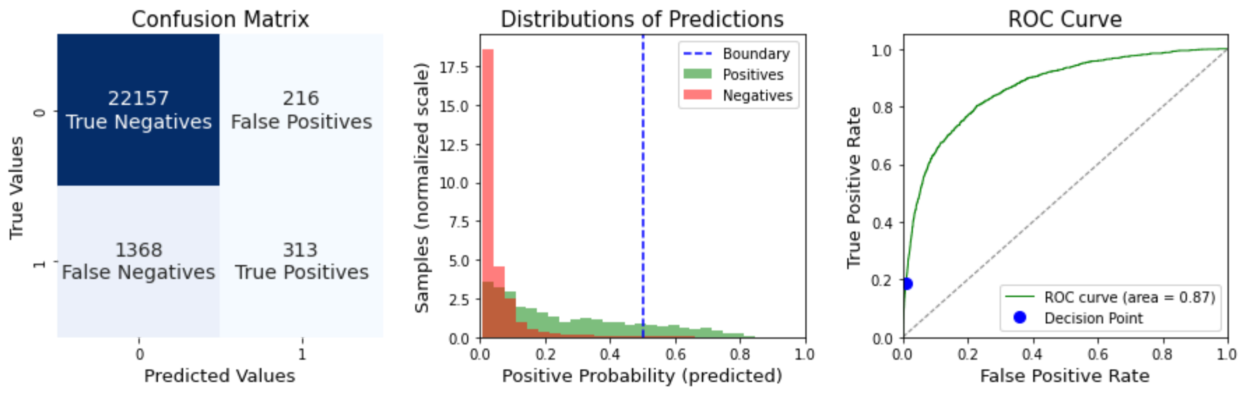
We fit the **train_x** and **train_y** obtained from the **train_test_split** to the **XGBClassifier** using 220 **n_estimators** and now predict the class probabilities of **val_x**, which we call **y_scorexgb**. We now get the **ROC AUC** score of 0.8649623562309385 using **val_y** as the actual values of y and **y_scorexgb** as the predicted y and plot the ROC curve:



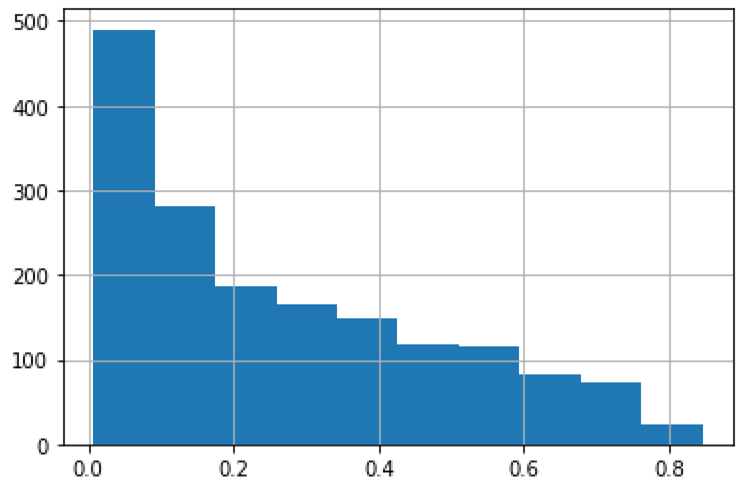
We perform a similar process with the XGBClassifier, this time with 120 n_estimators and predicting the class probabilities of **val_x**, which we now call **y_score2**. We now get the **ROC AUC** score of 0.8650872730959465 using **val_y** as the actual values of y and **y_score2** as the predicted probabilities of y and plot the **ROC** curve:



We now use the **XGB** model, **val_x**, and **val_y** as the parameters for **evalBinaryClassifier**, which gives us:



We now create a dataframe that shows which people have been labeled 0 (will not experience financial distress in 2 years) or 1 (will experience financial distress in 2 years), and the probabilities of being given this label. We filter the dataframe by those who have been labeled as 1 and plot the probability distribution:



From the distribution we can see that for many people, the probability of being labeled as 1 is very low. After finally completing the training, using the **XGBoost** classifier model, we predict the probability of being labeled as 1 using the testing data and place this information in the **sampleEntry** csv file.

1.4 Outcomes

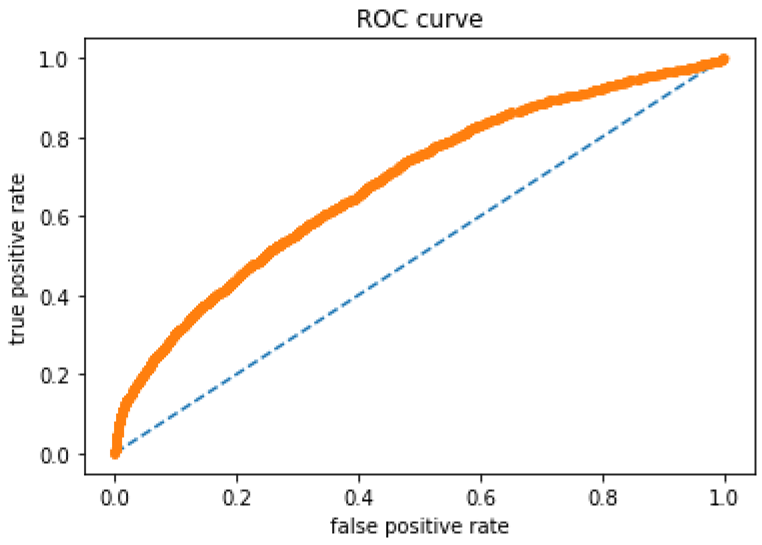
In order to analyze the effectiveness of the ADS, we will compare its performance across different subpopulations by creating these subpopulations from the already established variables (e.g. we will potentially create subpopulations based on monthly income: one population will have a lower income, another will be average income, and the third will be high income). We will likely quantify the individual and group fairness of this ADS based on the subpopulations we will create. Any additional methods for analyzing the performance of this ADS will most likely be added as we continue working on this project.

For this section, we will be focusing entirely on the second solution. According to the methodology of the ADS, no features appear to be excluded in the feature selection. In order to better understand the influence of certain features on the predictions, we thought it would be better to use a large sample size, since a larger sample size could potentially improve the ADS's performance. Instead of taking the author's approach of dropping the null values, we decided to impute the null values with the respective median value of each column/feature. We wanted to evaluate the model's fairness with respect to monthly income, which is a non-binary feature. As a result, we created a feature called **binary_income** that would allow us to classify people as having a monthly income of 1 (higher monthly income than the median monthly income) or 0 (a monthly income that is lower than or equal to the median monthly income). So, the shape of the training dataset we used is **(150000, 12)**.

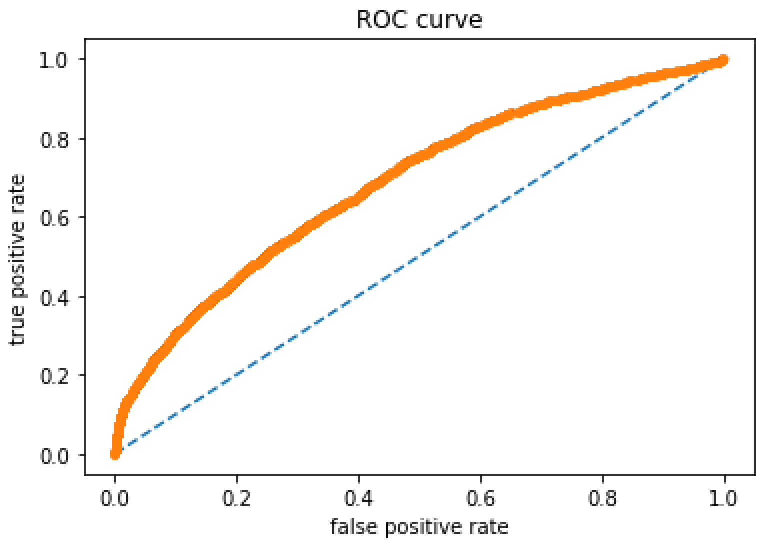
We utilized the same training process used by the author. We decided to focus on disparate impact as a measure of fairness since we're evaluating the model's fairness with respect to groups of people with higher income (**privileged_groups**) and groups of people with lower income (**unprivileged_groups**), meaning that we are dealing with group fairness, which is what disparate impact is meant to solve. After imputing the null values with the medians, we created the **binary_income** column and input the appropriate values. We then transformed the training data into a **BinaryLabelDataset**, setting the **binary_income** as the **protected attribute** and the **SeriousDlqin2yrs** as the **target**.

After plotting the importance of the features, we convert the training **BinaryLabelDataset** to a dataframe in order to plot its ROC curve and put it through the **logistic** function in order to predict and model y by using the **LogisticRegression()** classifier.

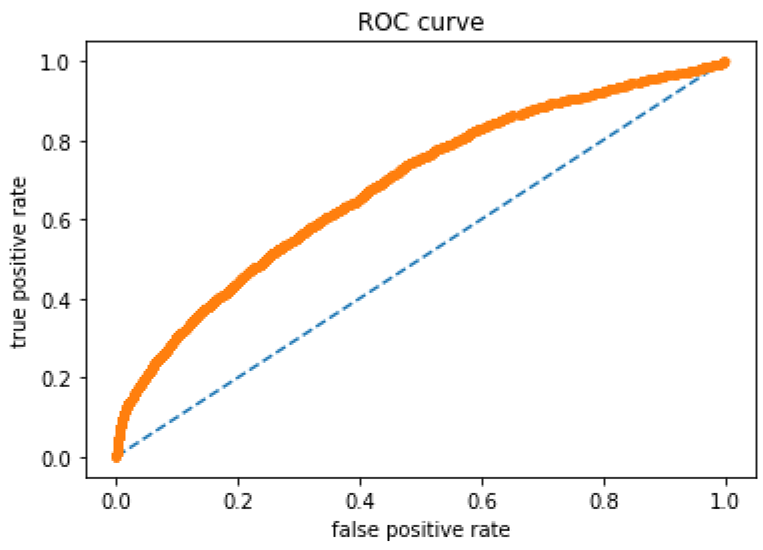
Using the class probabilities as **probs**, and the predicted classes as **y_test**, the function then returns the **ROC AUC score** of 0.6866285147099415, implying better accuracy than the author's solution. We then plot the **ROC** curve using **probs** and **y_test**:



We perform a similar process again, performing **train_test_split** on the same x and y we split previously, fitting x and y to the logistic regression model. We now predict the actual classes using **val_x** resulting in the predictions of y as **pred_y**. We now get the **ROC AUC score** of 0.5077903948364815, once again implying higher accuracy than the author's solution, using **val_y** as the actual values of y and **pred_y** as the predicted y and plot the **ROC** curve:

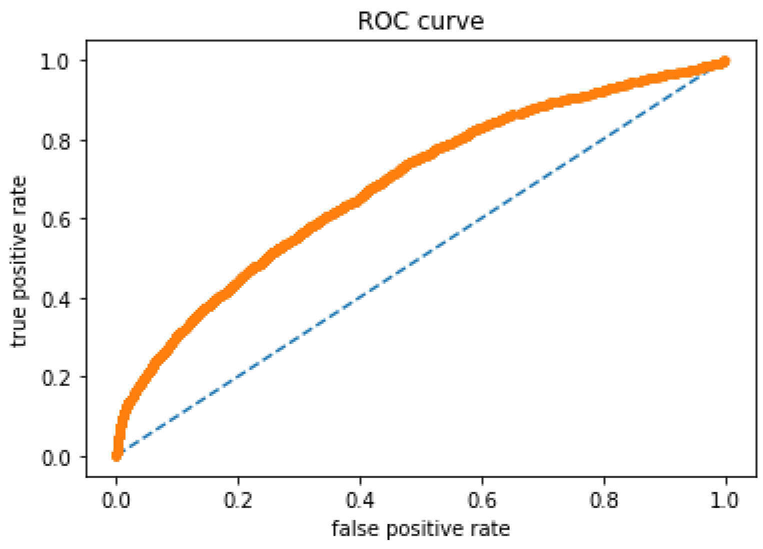


We fit the **train_x** and **train_y** obtained from the **train_test_split** to the **XGBClassifier** using 220 n_estimators and now predict the class probabilities of **val_x**, which we call **y_scorexgb**. We now get the **ROC AUC** score of 0.8653919879121579, implying higher accuracy once again, using **val_y** as the actual values of y and **y_scorexgb** as the predicted y and plot the **ROC** curve:

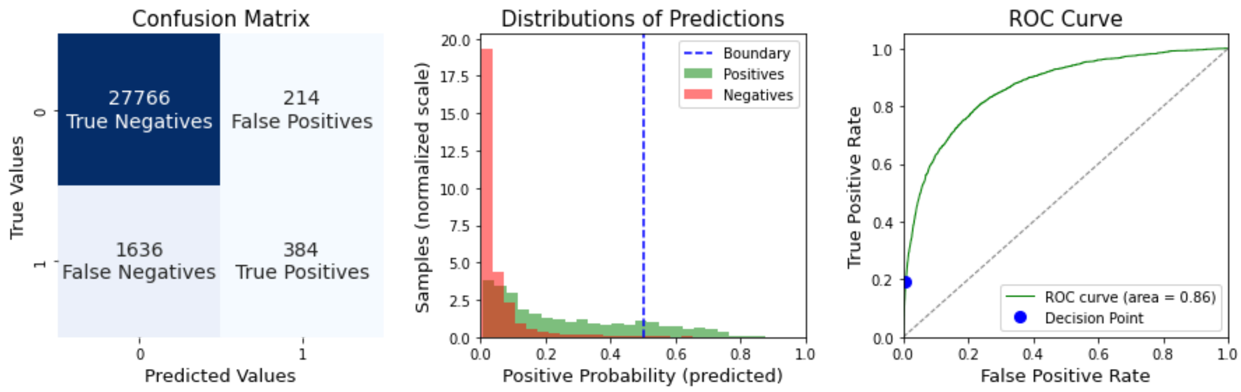


We now get the **BinaryLabelDatasetMetric** of the training **BinaryLabelDataset**, and get a disparate impact of 1.416363 implying that the outcomes are biased towards the **unprivileged_groups**. In the original implementation, we tried to find the disparate impact by performing a similar process except we dropped the null values, giving us a disparate impact of 1.597059, implying higher bias

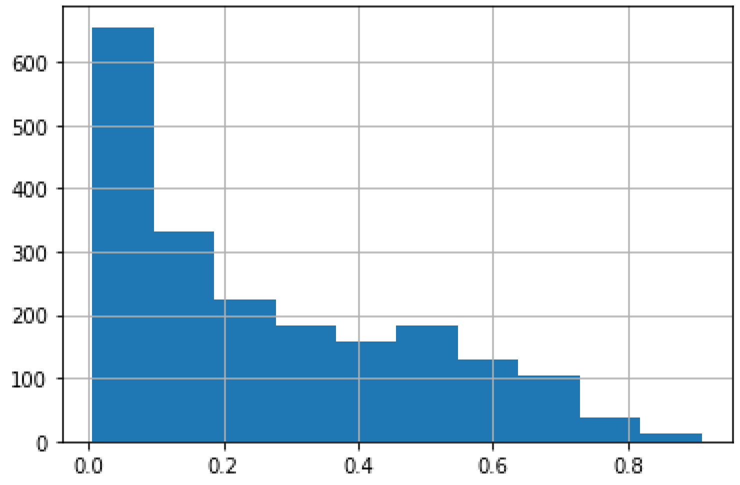
We once again get the **ROC AUC** score, this time at 0.8644105319216696 using **val_y** as the actual values of y and **y_score2** as the predicted y and plot the **ROC** curve:



And input **xgb_cfl2**, **val_x**, **val_y** into the **evalBinaryClassifier**:



We plot the probability distribution of the dataframe containing the data that classifies who will be in financial distress and the probabilities of these classifications:



After completing the training, we then decided to impute the null values of the testing data, putting it through a process similar to the one conducted on the training data, and then proceeding to use the **XGBoost** classifier model, we predict the probability of being labeled as 1 using the testing data and place this information in the **sampleEntry** csv file.

To get the disparate impact of the testing data, we converted it to a **BinaryLabelDataset**, however, in order to do so, we had to get rid of the null values, which we did by replacing the null values with **0**, since we wanted to change the testing data as little as possible. Using the **BinaryLabelDatasetMetric**, we calculate a disparate impact of 2.011160 for the testing data. Afterwards, we transformed the training data and testing data to **AI Fairness 360** (**aif360**) and called on the **DisparateImpactRemover** at different levels, including: **0**, **0.25**, **0.5**, **0.75**, **1**, and completed the following process five times for each level. We used **fit_transform** at each level for both the training and testing **aif360s**. We also used **fit_transform** on the **aif360s** features, and converted them to dataframes to drop the **target** and **protected_attr** to obtain x for the training and testing dataframes. We obtained y for the training and testing dataframes by getting their columns for **target**. We fit each level's **train_x** and **train_y** to the **XGB Classifier** and calculated each of their **y_scorexgb**, which are essentially the predicted probabilities of being labeled as 1 using the testing data's x. Using **predict_prob** on the testing x, we also calculate each of their **pred_y**, which are essentially the predicted labels themselves. We then calculate their **disparate impact**, **mean difference**, **ROC AUC scores**, and **overall accuracies**:

Repair Levels	**Mean Difference**	**Disparate Impact**	**ROC AUC Score**	**Overall Accuracy**
0	0.012357	2.017640	1.0	0.9963449356176665
0.25	0.010353	1.800785	1.0	0.9960690817020187
0.5	0.009270	1.688208	1.0	0.9955666334985173
0.75	0.006145	1.393704	1.0	0.9941775119947194
1	0.005496	1.347220	1.0	0.993478025280041

It is important to note that these values may not be particularly correct. In order to calculate the disparate impact of the testing data, we had to be rid of the null values, and by inputting **0**, it is possible that the calculations we did after using the **DisparateImpactRemover** were affected.

1.5 Summary

Overall, we don't believe the data is appropriate for the ADS. After replacing the null values with the values' medians, the ADS became more accurate - however, the **disparate impact** (which was about **1.42**) **indicated more positive outcomes for the unprivileged group**, **implying that they were overall more likely to be labeled as someone who would experience financial distress in two years**, therefore indicating incredible bias against the unprivileged group. This is likely because after replacing the null values, the number of individuals in the **privileged group** was **59954**, while the number of individuals in the **unprivileged group** was **90046**. This is a significant difference to the numbers in the original solution, which gave us **59954** for the **privileged group** and **60315** for the **unprivileged group** after dropping the null values, indicating that although **there was less accuracy in the original solution, there was also likely less bias against the unprivileged group** since there was less data about them.
If such an ADS's predictions were utilized to make decisions (such as banks determining whose loans to approve based on the likelihood of being paid back), it would further encourage the idea

that those with lower income are more likely to remain financially distressed. However, this might simply be a case of **pre-existing bias**, since it is not unusual for lower income individuals to remain financially distressed due to not having the access to certain resources such as higher education, which could potentially lead to landing a job with higher income. In addition, if the data was taken from individuals during a time where the economy was deteriorating, it could lead to bias against those whose income is low due to economic circumstances beyond their control.

One of the issues with the original implementation is that it left out many individuals by dropping the null values, making the sample much smaller and therefore decreasing the accuracy of its predictions. By inputting the null values, we were able to utilize a larger sample, as well as increase the ADS's accuracy. In addition, we created a new feature that would allow us to better analyze the ADS's fairness with respect to people's income. As mentioned previously, this new feature called **binary_income** would allow us to label **whether an individual had a high income and was therefore privileged, or had a low income, making them unprivileged**. It would also allow us to calculate the **disparate impact** of the model, which we already explained was **incredibly biased**. Even though we tried to use the **disparate impact remover**, somehow the **disparate impact worsened/showed little improvement, despite the ROC AUC scores and overall accuracy scores indicating "perfect" accuracy**. However, because of the significant amount of bias within the data, we are reluctant to agree with these high accuracy scores, since the predictions would be unfair towards certain groups.

In order to improve upon this ADS, one could either use a much larger sample size and drop the null values, or use a larger sample size with less null values- although the latter would be optimal, it would likely be more difficult to acquire. In addition, it would be best to remove the **disparate impact** as soon as possible in order to mitigate bias (**such as removing the disparate impact from the training data right at the beginning**).Due to certain limitations and judging from the **accuracies** and **ROC AUC scores**, we may have been unable to completely and properly mitigate biases. However, it seems we were able to optimize accuracy.

Such a high accuracy would likely benefit banks by allowing them to correctly determine which borrowers should be given loans, as well as benefit borrowers by giving them information that could help them improve their financial habits. A lower disparate impact would benefit lower income borrowers, since it would help them be treated more fairly by the banks; a lower disparate impact would also benefit banks since it would prevent them from discriminating against potential clients, therefore potentially increasing their credibility and attracting more clients. As a result, **we believe that optimizing both the accuracy and disparate impact of the ADS would be most beneficial for banks and borrowers alike**.