

Machine Learning Engineer Nanodegree

Capstone Project

Éderson André de Souza

October 12, 2019

I. Definition

Project Overview

It is not unusual to hear a company's management speak about forecasts: "Our sales did not meet the forecasted numbers," or "we feel confident in the forecasted economic growth and expect to exceed our targets." You cannot predict the future of your business, but you can reduce risk by eliminating the guesswork. With accurate forecasting, you can make a systematic attempt to understand future performance. This will allow you to make better informed decisions and become more resistant to unforeseen financial requirements. Without correctly estimating financial requirements and understanding changing markets, your business decisions will be guess work which can result in insufferable damage.

So, with that in mind, without doubt, it is very important to help business forecasting future products and services demands.

And because of that, I chose the Santander Customer Transaction Prediction dataset to try building a model that can consistently handle this task.

In this project, I trained and tested a binary classifier capable of predicting the probability of a customer make a specific transaction in the future. The model used the gradient boosting algorithm, and was trained on the data provided for [their Kaggle competition](#).

Problem Statement

Banco Santander, S.A., doing business as Santander Group, is a Spanish multinational commercial bank and financial services company based in Madrid and Santander in Spain. Additionally, Santander maintains a presence in all global financial centres as the 16th-largest banking institution in the world. Although known for its European banking operations, it has extended operations across North and South America, and more recently in continental Asia. [Wikipedia](#)

In their [Kaggle competition](#), Santander provided an anonymized dataset containing numeric feature variables, the binary target column, and a string ID_code column; the goal is to build a model that predicts the probability of a customer make a specific transaction in the future.

To solve this problem, firstly I attained some basic understanding of the training data that was provided to check for characteristics such as target concept balance and distribution of features; Then, I used a classification model based on the well known Logistic Regression as my baseline model for comparison. The final model is build on a gradient boosting framework, The XGBoost. I

also have decided to evaluate the amount of data used, and performed parameter tuning for the GB model. Lastly, I tested the final model design using its performance on the given testing data.

Metrics

Following [Kaggle's competition metrics](#), a model is graded based on the area-under-the-ROC-curve score between the predicted class probability and the observed target, measured on the test data. Since the test data is not labelled, grading is done by uploading the file containing the probability of each transaction in the test data to Kaggle.

ROC (Receiver Operating characteristic) curve is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters:

- True Positive rate
- False Positive rate

AUC stands for "Area under the ROC Curve." That is, AUC measures the entire two-dimensional area underneath the entire ROC curve (think integral calculus) from (0,0) to (1,1).

source: [Google](#)

Using the ROC-AUC metric, we can see the closeness between the model's prediction on the testing set and that of the solution, showing how accurate the model is in terms of classification probability. Furthermore, ROC-AUC can tell how good the classifier is at separating the positive and negative. While , ROC-AUC is a quick and intuitive way of assessing the model's performance compared to the solution.

In my opinion, another reason why ROC-AUC was chosen to measure classifier performance in this competition instead of for example simple accuracy metric, is because of its sensitivity towards True Positive and False Positive. As you may notice, accuracy don't work well for imbalanced datasets.

II. Analysis

Data Exploration

The dataset provided by Santander on [Kaggle competition](#) includes approximately 400 thousand customers, split equally into training and testing sets. So, the training set contains 200 thousand rows of data, each has 200 numbered features (columns) without any description, an ID_code column and a binary target, which is whether that customer did a transaction or not. The testing set contains the same amount of data, 200 thousand rows, and the main difference between them is that in the test set there is no target.

Files:

- train.csv (288.33 MB): the training data
- test.csv (287.75 MB): the test data
- sample_submission.csv (2822 KB): submission model

Line counts:

- train.csv: 200,000
- test.csv: 200,000

Feature Descriptions:

- On both sets:
 - `var_0` to `var_199` (float): no description about them.
 - `ID_code` (string): row identification.
- Training set:
 - `target`: whether customer did a transaction or not; 1 - yes, 0 - no

Observations:

- The data is very imbalanced with only 10.049% positive (customers that made a transaction). And because of that *Accuracy Score* doesn't work well here. If we guess that all customers won't make the transaction, our *Accuracy* would be 89.951%, pretty impressive, no? **Not at all.**
- All features were previously normally distributed.
- There aren't missing values in the data.
- Basic Statistics:
 - *Mean*: The Mean of the values stays between -16.55 -> 24.52.
 - *Median*: Because the distribution of the features was normalized, the Median range is almost like the Mean: -16.48 -> 24.45.
 - *Standard Deviation*: 0.01 -> 21.40.
 - *Min*: Range -90.25 -> 13.73
 - *Max*: Range 1.00 -> 74.03
- There is no strong correlation between features.
 - Strongest Positive Correlation: 0.0097
 - Strongest Negative Correlation: -0.0098

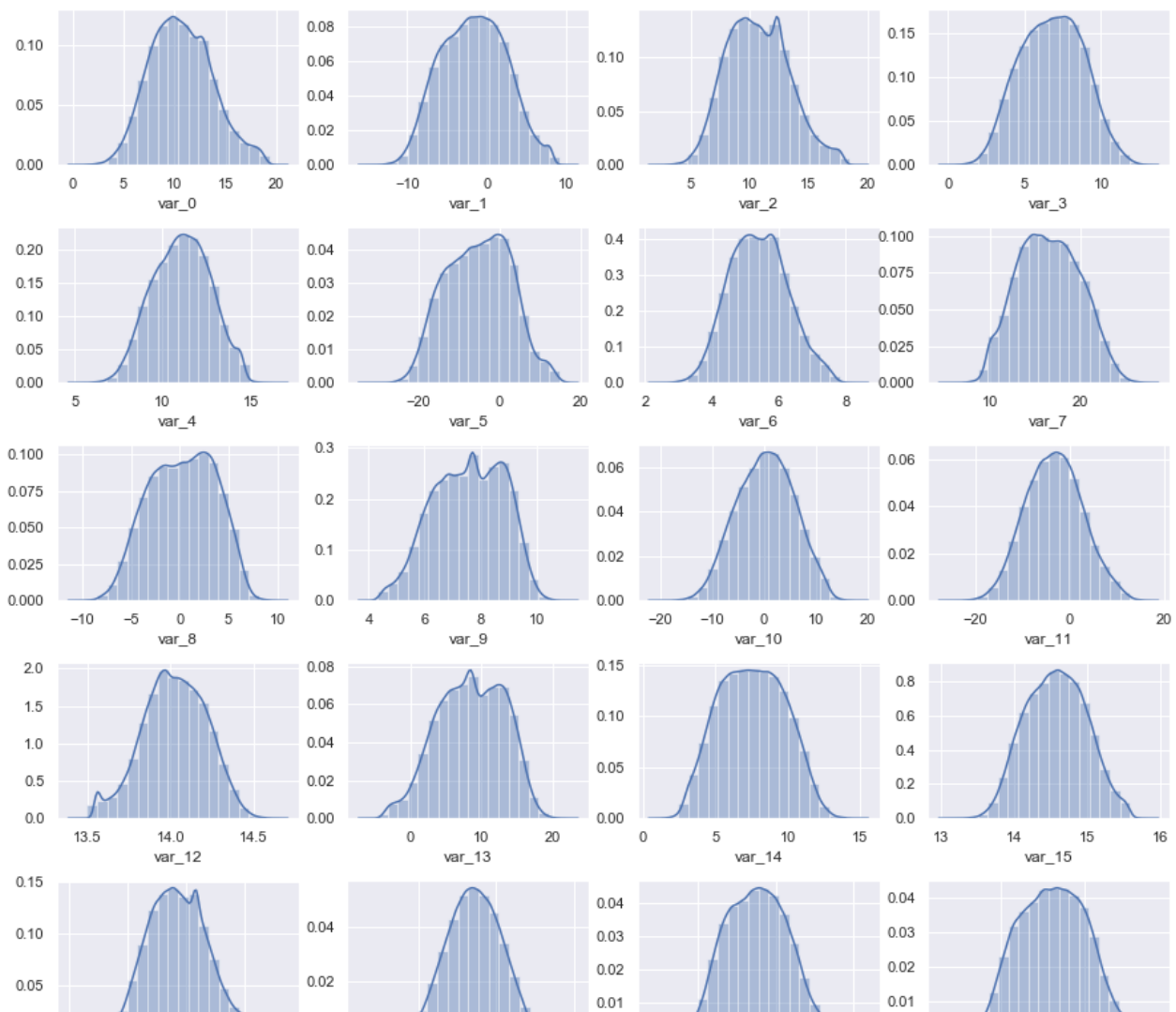
Exploratory Visualization

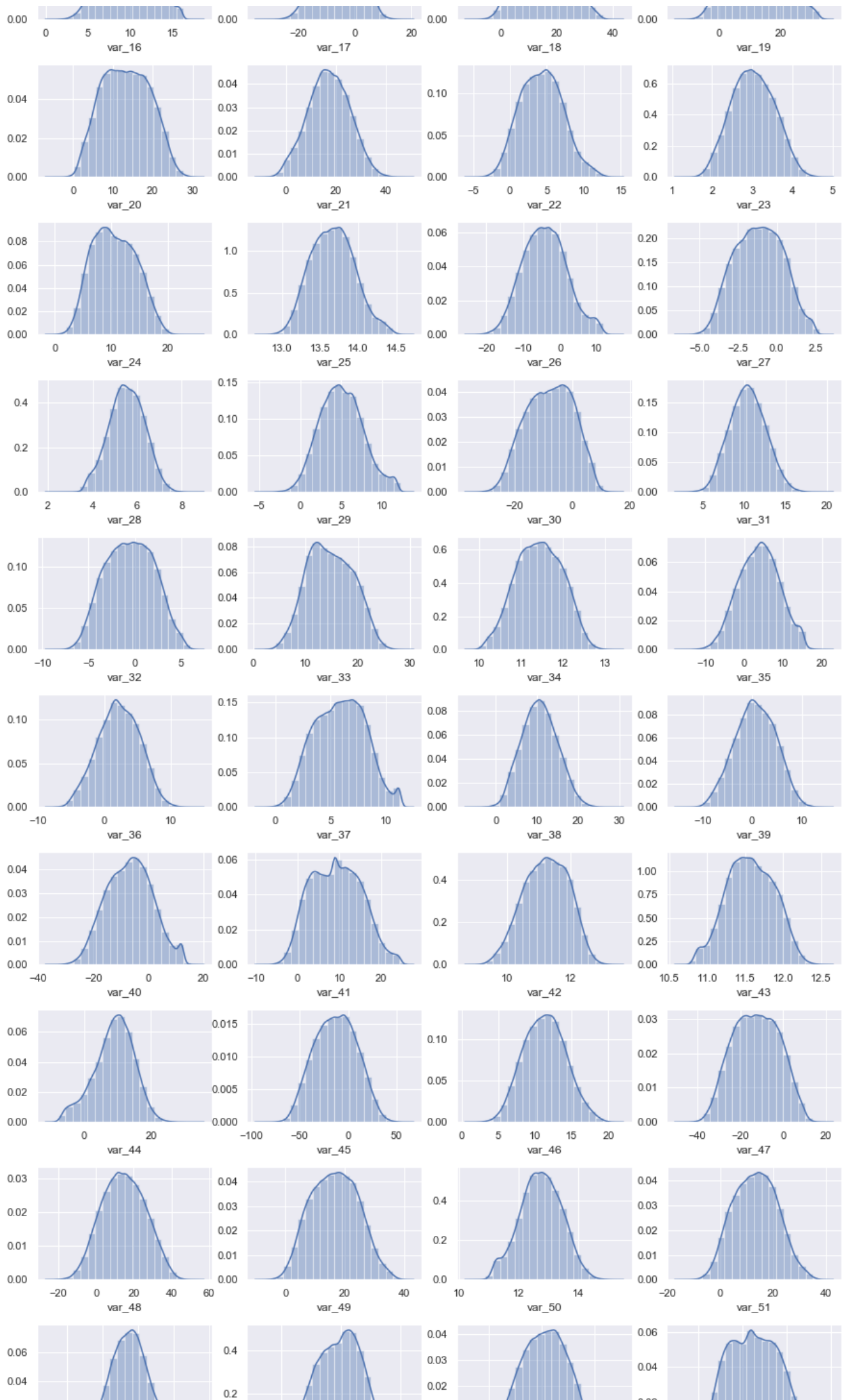
The plot below shows the distribution of the target in the training set:

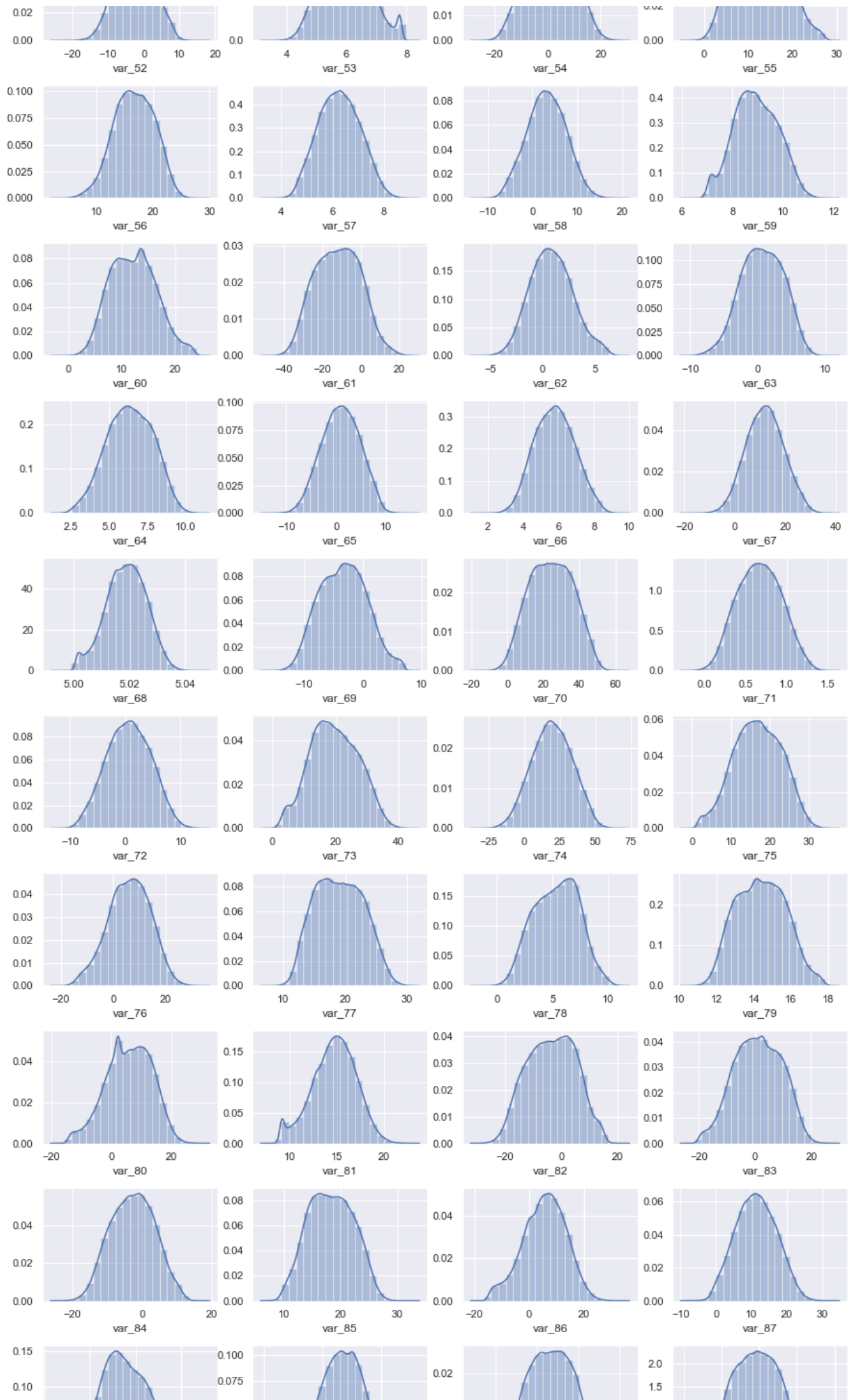


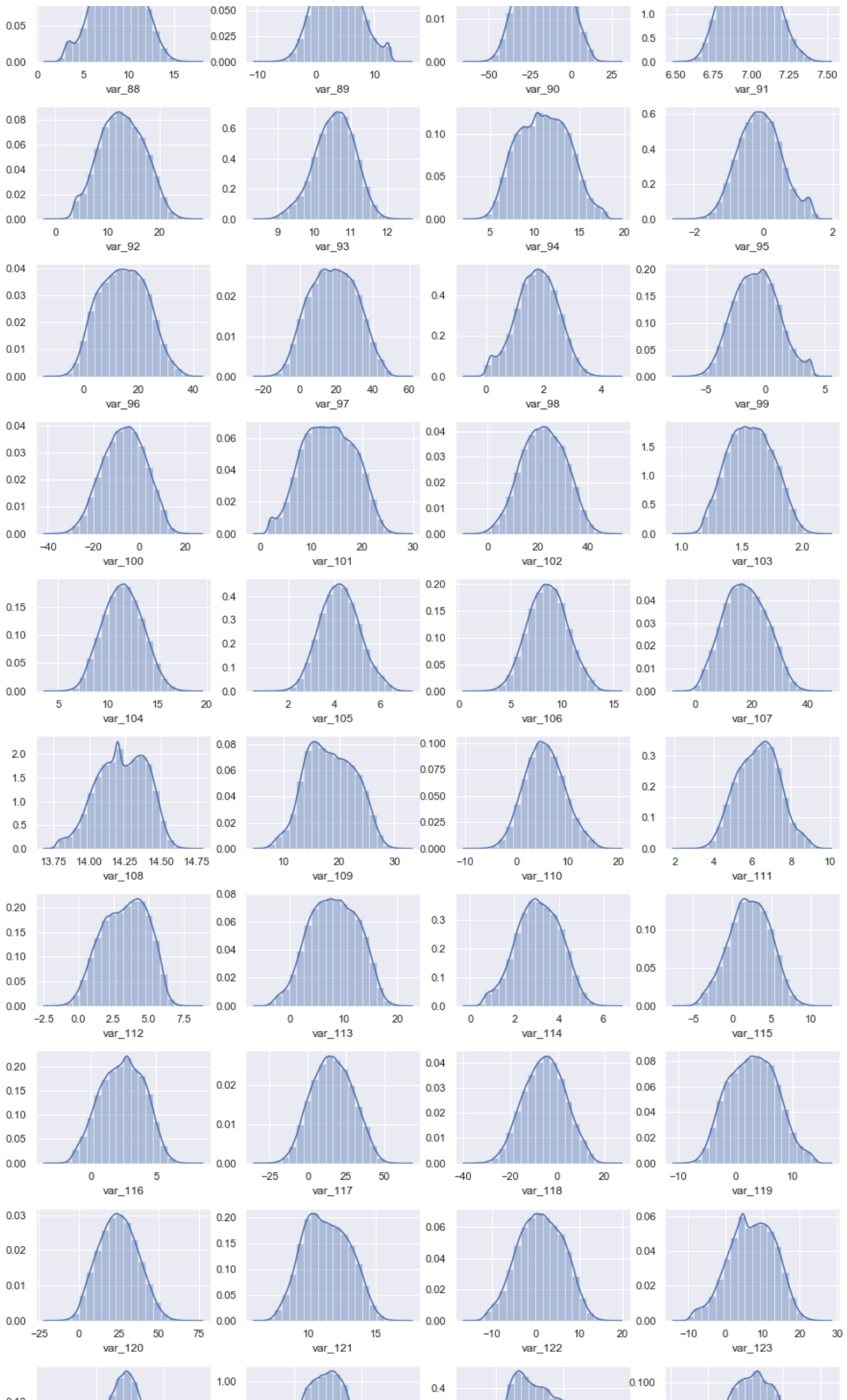
This may need to be taken into consideration later when the classification model is chosen.

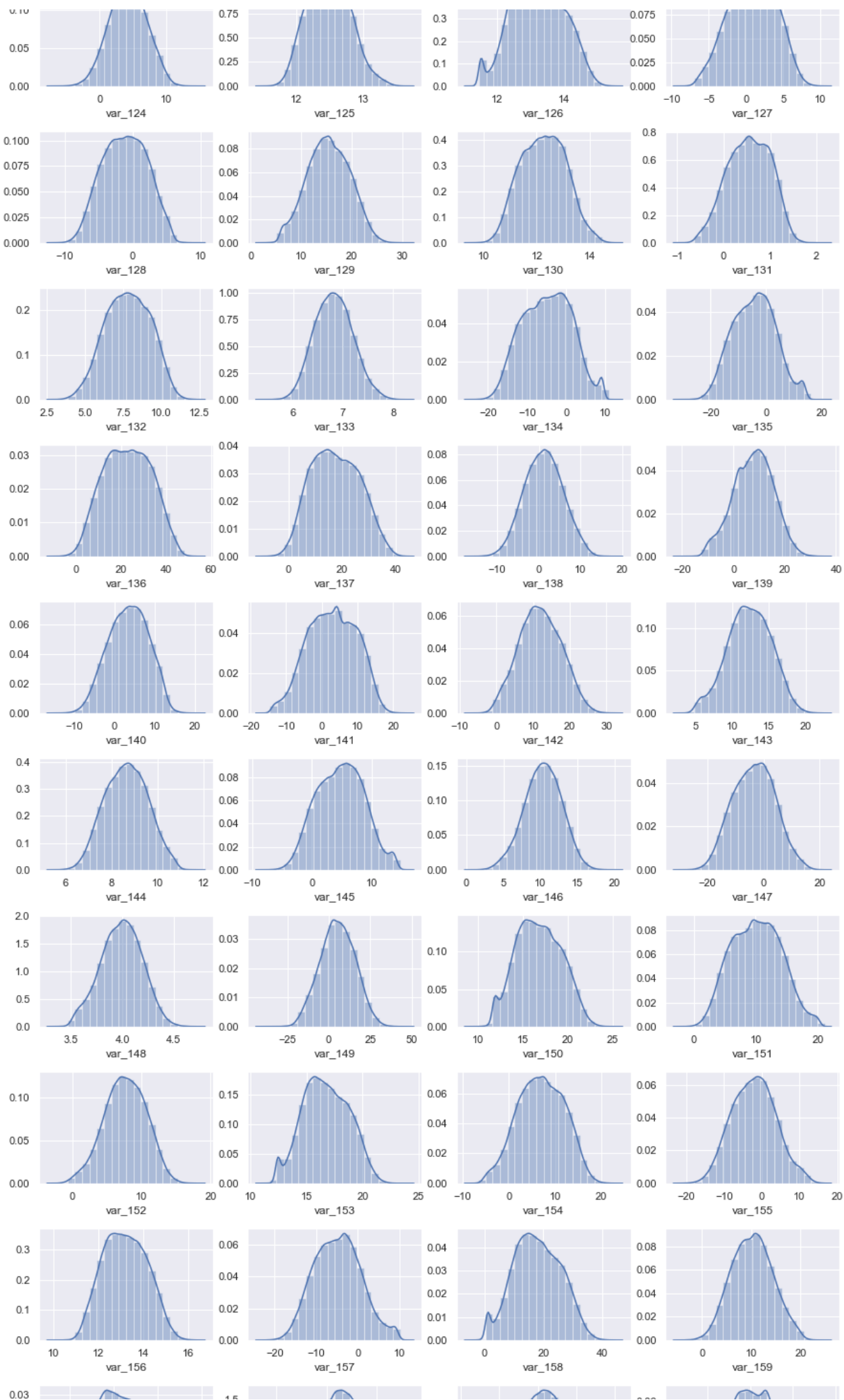
Let's visualize the feature distributions:

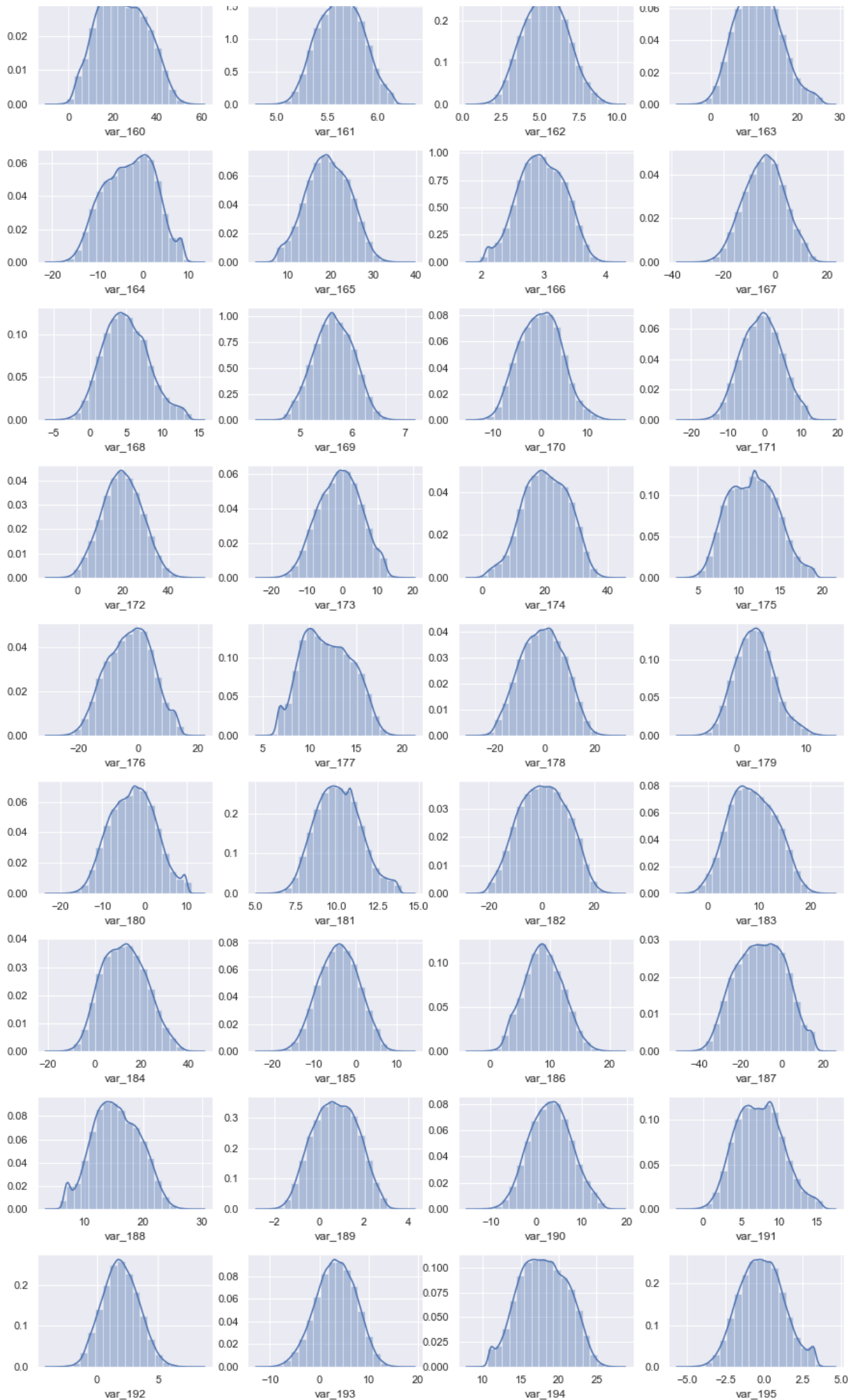


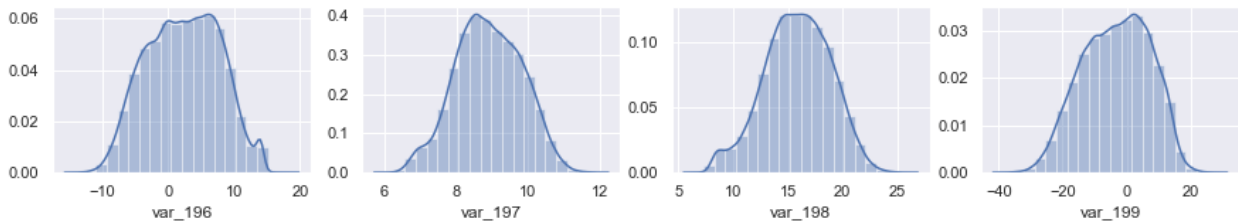












As mentioned above, all features show distributions like Normal.

Algorithms and Techniques

Extreme Gradient Boosting (XGBoost) is an implementation of the gradient boosting machines that is highly flexible and versatile while being scalable and fast. XGBoost works with most regression, classification, and ranking problems as well as other objective functions; the framework also gained its popularity in recent years because of its compatibility with most platforms and distributed solutions like Amazon AWS, Apache Hadoop, Spark among others.

In short, XGBoost is a variation of boosting - an ensemble method algorithm that tries to fit the data by using a number of "weak" models, typically decision trees. The idea is that a "weak" classifier which only performs slightly better than random guessing can be improved ("boosted") into a "stronger" one that is arbitrarily more accurate. Building on the weak learners sequentially, at every round each learner aims to reduce the bias of the whole ensemble of learners, thus the weaker learners eventually combined into a powerful model. This idea gave birth to various boosting algorithms such as AdaBoost, Gradient Tree Boosting, etc.

XGBoost is an example of gradient boosting model, which is built in stages just like any other boosting method. In gradient boosting, weak learners are generalized by optimizing an arbitrary loss function using its gradient.

XGBoost, as a variation of boosting, features a novel tree learning algorithm for handling sparse data; a theoretically justified weighted quantile sketch procedure enables handling instance weights in approximate tree learning. [Reference 01](#) [Reference 02](#) [Reference 03](#)

There is a number of advantages in using XGBoost over other classification methods:

- **Work with large data:** XGBoost packs many advantageous features to facilitate working with data of enormous size that typically can't fit into the system's memory such as distributed or cloud computing. It is also implemented with automatic handling of missing data (sparse) and allows continuation of training, or batch training.
- **Built-in regularization:** XGBoost supports several options when it comes to controlling regularization and keeping the model from overfitting, including gamma (minimum loss reduction to split a node further), L1 and L2 regularizations, maximum tree depth, minimum sum of weights of all observations required in a child, etc.
- **Optimization for both speed and performance:** XGBoost provides options to reduce computation time while keeping model accuracy using parallelization with multi-core CPU, cache optimization, and GPU mode that makes use of the graphics unit for tree training.

Benchmark

For the baseline benchmark, I have randomly predicted with 10% probability (the distribution of the training set) that a customer will make a transaction. This method yields an AUC score of ~ 0.50 on the submission to Kaggle. This is equivalent to guess that all customers will not make the transaction, which is very bad and naive.

So, if the final model results in an AUC score better than the 0.50, we have succeeded.

Because of this naive approach, I have decided to bring a Baseline Model for benchmark too. It is Logistic Regression, one of the simplest classification models in the toolbox of any Data Scientist. This problem has a binary target, so Logistic Regression can be applied. LR is easy to implement, tune, update and interpret.

[Reference 01](#) [Reference 02](#) [Reference 03](#) [Reference 04](#)

So, in the end, our main goal is to beat the Logistic Regression Model AUC, which is 0.8603 on the validation subset, using about 50% of the dataset (we will discuss sample size in the next section).

Just for clarification, the train dataset was split into train and validation subsets. The train subset got 80% of the data and validation got the 20% left.

III. Methodology

Data Preprocessing

As partially mentioned, the given dataset had already been well-prepared and processed, all categorical features had been labelled and set to binary numbers, features normalized, and potentially missing data had been either filled or discarded.

Feature engineering

In my opinion, the hardest task in this project was feature engineering.

The main reason is that there is no explanation about the features. All of them are meaningless numbers and without correlation. And for leaving this task even harder, there are 200 features, which demand a lot of computational power to work with. So, I didn't make any feature engineering in this project.

Feature Scaling and Data Sampling

As you may notice at Data Exploration, each feature has its own range of values, and good practice is to bring all features to the same level of magnitudes with Feature Scaling.

There are many techniques to handle this job like Rescaling (min-max normalization), Mean normalization, Standardization, Scaling to unit length and others.

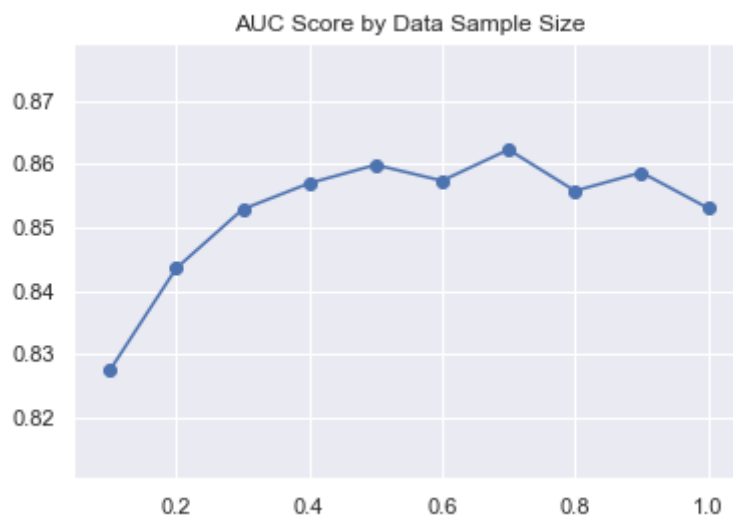
However, I have chosen Standardization, which makes the values of each feature in the data have zero-mean and unit-variance. The main reason for choosing Standardization is because this method is widely used in many machine learning algorithms and it is very effective for Logistic Regression.

For Tree-based models like XGBoost, Feature Scaling is not required, but I have used to try speed up the calculations.

[Reference 01](#) [Reference 02](#)

In order to save computational power and time, I have tested the baseline model with Stratified Samples of the training set. The sample size started at 10% and was incremented by 10% every test until 100%, meaning the entire dataset.

Plotting results obtained, we can see that the best result (AUC = 0.8623) came from 70% of the dataset size and using only 50% of it also shown a very good result (AUC = 0.8599).



Another big advantage of using a stratified sample of the data was time-saving. Training with 70% of the dataset saved about 35% of the time, and 50% of the dataset saved about 55% of the time to train the model.

Given that, the final model will be trained using only 50% of the dataset.

Handling Imbalanced Dataset

As we saw earlier, the training data set is very imbalanced. Only 10.049% of the target is positive. An approach to this problem is oversampling. Oversampling can be defined as adding more copies of the minority class based upon the existing observations. [Reference](#)

In this case, I have applied a technique called SMOTE. What it does is: *"First it finds the n-nearest neighbors in the minority class for each of the samples in the class . Then it draws a line between the the neighbors an generates random points on the lines"*. [Reference](#)

Applying SMOTE on the Baseline Model, it was not observed any improvement on AUC Score, but the big impact was on Recall Score. It jumped from 0.246854 to 0.769603, which means that 76.62% of the positive targets were detected. Therefore, in case Santander needs to predict all transactions as possible, the higher the Recall Score is, the better.

At the XGBoost final model, the oversampling technique is already inside the hyperparameters (scale_pos_weight). Therefore, SMOTE package won't be used.

Implementation

The software requirement for the implementation is as followed:

- Python ≥ 3.7
- numpy $\geq 1.16.5$
- pandas $\geq 0.25.1$
- scikit-learn $\geq 0.21.3$
- xgboost ≥ 0.90
- hyperopt ≥ 0.2

I first attempted to train an XGBoost model right of the box, without tuning. Then, I planned to compare the performance to the same model trained with hyperparameter tuning. In the first attempt, the only hyperparameter passed was `n_jobs = -1` to force using all CPU cores.

The AUC Score obtained was 0.8306 on the validation subset, which is not good compared with our baseline model, Logistic Regression.

Let's tune the model to see if we can get a better result.

Refinement

Hyperparameter tuning

The refinement process started with hyperparameter tuning. To handle this task I have decided to use Hyperopt package.

Hyperopt uses Bayesian Optimization which combines randomness and posterior probability distribution in searching the optimal parameters. In contrast, GridSearch from SKLearn uses purely random methodology, which is not the smart way. [Reference](#)

The optimized parameters were:

- `gamma`:
 - Minimum loss reduce for each node split
 - Search values: [5..10]
- `min_child_weight`:
 - Minimum sum of weights of all observations required in a child node
 - Used to avoid overfitting as higher values prevent model to learn relations specific to the dataset
 - Search values: [1..5]
- `subsample`:
 - Subsample ratio of the training instances
 - Search values: [0.5..1.0]
- `colsample_bytree`:
 - Control the amount of features to be sampled by each tree
 - Search values: [0.2, 0.9]
- `reg_alpha`:
 - L1 regularization
 - Search values: [0.1..1.0]

- `scale_pos_weight`:
 - Control the balance of positive and negative weights, useful for unbalanced classes.
 - Search values: [8..10]

Running 500 evaluations the best parameters found were:

- `colsample_bytree`: 0.2
- `gamma`: 9.0
- `min_child_weight`: 4.0
- `reg_alpha`: 0.35
- `scale_pos_weight`: 8.920569929381536,
- `subsample`: 0.55

For the final model it was added the following parameters:

- `learning_rate`: 0.2
 - Step size shrinkage used in update to prevents overfitting.
- `max_depth`: 2
 - Maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit.
- `objective`: "binary:logistic"
 - Logistic Regression for binary classification, output probability
- `tree_method`: 'gpu_hist'
 - Fast histogram optimized approximate greedy algorithm. It uses some performance improvements such as bins caching.
 - GPU acceleration.
- `verbosity`: 0
 - Verbosity of printing messages.

IV. Results

Model Evaluation and Validation

With the model properly tuned, it is ready to finally be trained.

To assure the results and evaluate the robustness and generalization, one best practice is to use K-Fold Cross Validation (CV).

Usually, we split the data set into training and testing sets (as we did) and use the training set to train the model and testing set to test the model. Then, we evaluate the model based on an error metric to determine the accuracy of the model. This method however, is not very reliable as the accuracy obtained for one test set can be very different to the accuracy obtained for a different test set. K-fold Cross Validation (CV) provides a solution to this problem by dividing the data into folds and ensuring that each fold is used as a testing set at some point. [Reference](#)

I have used `RepeatedStratifiedKFold` from SKLearn Library to apply Cross Validation. The parameters passed was:

- `n_splits` = 4
 - Number of folds or buckets.

- 4 buckets of data: 3 for training, 1 for testing.
- `n_repeats = 2`
 - Number of times cross-validator needs to be repeated.

In the end, the model was trained 08 times and the results stored in a dataframe to evaluation.

As mentioned before, I have used just 50% of the original train set.

The final scores using validation subsets are these:

- **AUC Score:**
 - Min: 0.886400
 - Mean: 0.889550
 - Max: 0.891600
- **Precision:**
 - Min: 0.382974
 - Mean: 0.387747
 - Max: 0.396058
- **Recall:**
 - Min: 0.720524
 - Mean: 0.734544
 - Max: 0.741961

Justification

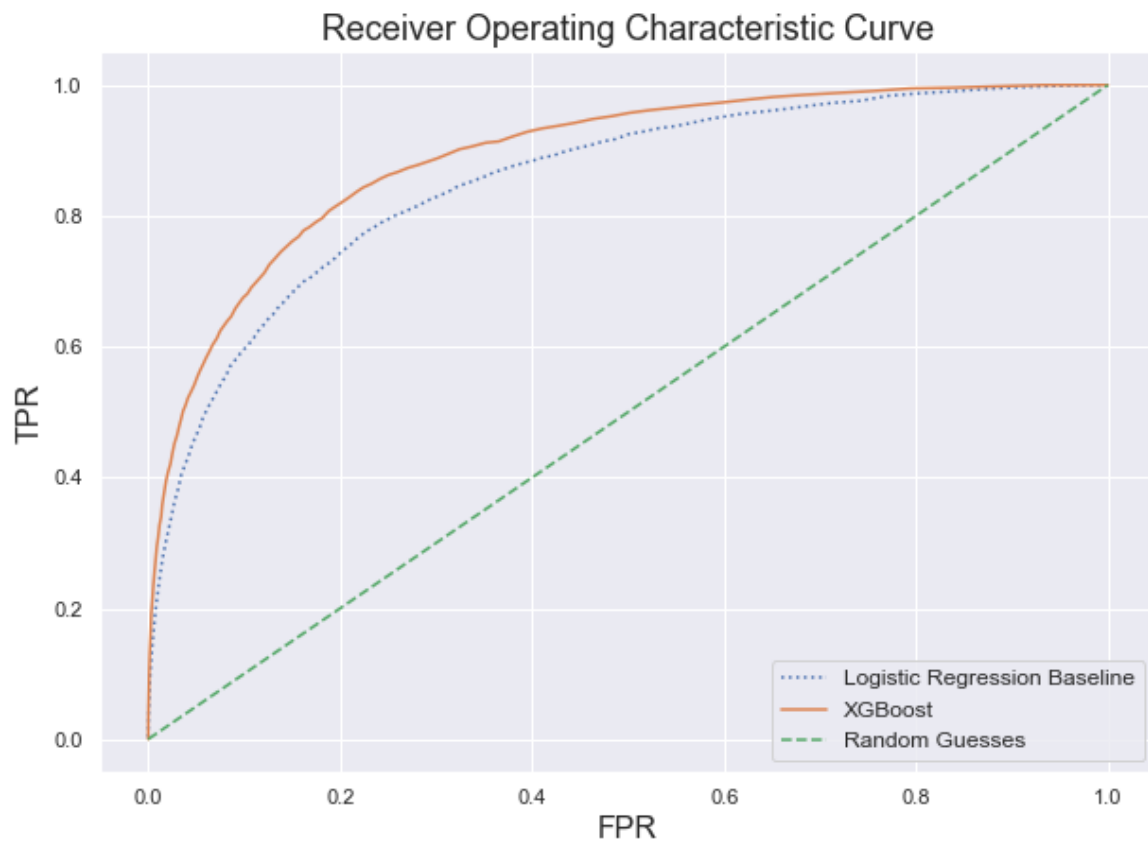
Analyzing the results above, we can see that tuned XGBoost got an average AUC Score near 0.89 using validation subsets, which is 3% over our baseline model (0.8599).

Looking at Precision Score, we notice about 8% of improvement over the baseline model. XGBoost gave us 0.3877, which means that of all positive predictions, XGBoost was correct 38.77% on average, while Logistic Regression 30,13%.

Recall, on another hand, shown a minor loss. Logistic Regression delivered a Recall Score of 0.7691, meaning that the baseline model detected about 76.91% of all True Positives. XGBoost, however, detected 73.45%.

Overall, in my opinion, XGBoost was better. We lose a little in Recall, but we gain in Precision, thus, XGBoost made fewer mistakes.

Let's visualize the Receiver Operating Characteristic Curve (ROC Curve):




Now it is time to make the submission to Kaggle website and see if our model really learned and will keep the AUC around 0.89.

Here is the result from Kaggle:

Featured Prediction Competition

Santander Customer Transaction Prediction

Can you identify who will make a transaction?

 Banco Santander · 8,802 teams · 6 months ago

\$65,000

Prize Money

Overview

Data

Notebooks

Discussion

Leaderboard

Rules

Team

My Submissions

Late Submission

Your most recent submission

Name	Submitted	Wait time	Execution time	Score
submission.csv	2 minutes ago	77 seconds	2 seconds	0.88997

Complete

As expected, we got **0.88997** very close to 0.89 from validation score. **Cheers!**

V. Conclusion

Reflection

End-to-end problem solution

This project turned out to be one that emphasizes on building and tuning model; the given data was very clean and thoroughly processed, the target concept is clearly defined, the features didn't give room for engineering. All that was left was working around the model, fine-tuning and searching for the best result.

Here is the pipeline solution:

- Establish basic statistics and understanding of the dataset such as imbalance, missing values, feature distributions, etc.
 - Data cleaning was not needed as the given data was thoroughly processed by Santander.
- Preprocessing Data:
 - Preprocess data using scaling and sampling.
 - Attempt to identify highly correlated features to drop, but there is none.
- Train and test model's performance:
 - Training Logistic Regression and XGBoost without tuning for benchmark comparison.
 - LR AUC Score: 0.8599
 - XGBoost AUC Score: 0.8306
- Improve:
 - Fine-tune model's parameters with **Hyperopt**
- Train and test again:
 - Kaggle submission AUC Score: 0.88997

Challenges

The major challenge encountered was working with all the features. I tried many different approaches to minimize the number of features, but in the end, the best result came from all of them together. Working in the dark, without knowing about the features was a little frustrating.

Improvement

There is so much that can be improved upon for this project.

I also have trained the XGBoost model with 100% of the data instead 50% and the result was 0.89603. Many tests were made and I was not able to reach 0.89993 like many competitors with this algorithm.

I would try to apply another algorithm like LightGBM and evaluate the performance.

If we look at the leaderboard from Kaggle, the best result is 0.92573, which, in my opinion, is not a giant difference from our final score, but 0.89603 still very far from the top at position 5167/8802.
