Santander Customer Transaction Prediction

Saurav Roy

October 25, 2019

Project-II

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Note: All code snippets and plots pertain to the codes used in Python. The results obtained in R are a little different to the results obtained in Python because the oversampling techniques for R and Python were slightly different. The split into train and test data also yielded different datasets, as they are randomly split.

CHAPTER I

INTRODUCTION

1.1. Background

The Santander Customer Transaction Prediction challenge is one that has plagued the industry for ages. It is a very common problem statement on various major online community platforms like Kaggle and various data scientists throughout the world have tried to run down machine learning algorithms in order to find the best solution possible.

The mission at Santander has always been to help people and businesses prosper. Their goal has been to help customers understand their financial health and identify which products and services might help them achieve their monetary goals. The data science team at Santander has always been trying to find out a solution for one of their most common challenges, namely binary classification problems such as: is a customer satisfied? Will a customer buy this product? Can a customer pay this loan?

1.2. Problem Statement

The goal of this challenge is to identify which customers will make a specific transaction in the future, irrespective of the amount of money transacted.

1.3. Datasets

Two datasets, namely *train* and *test* are provided for this task. Both datasets, consisting of 200,000 observations each, are anonymized and they contain 200 numeric feature variables (*var_0*, *var_1*, *var_2*... *var_199)*, and a string *ID_code* column. In addition, the train dataset contains a binary *target* column (*0* and *1*).

CHAPTER II

EXPLORATORY DATA ANALYSIS

2.1 Introduction

The very first thing to do before starting work on any dataset is to explore the data. This means a lot in the context of data analysis because looking at data not only means just visualizing the data in terms of graphs, but also cleaning the data and making it ready for analysis. This process is referred to as **exploratory data analysis** in the field of data analytics. To do this, we look at certain features present in the dataset and convert them, if necessary, to their proper forms which can be fed to the machine. Not only this, we also remove a few features which we feel may not contribute to the overall prediction process.

2.2. Removal and conversion of features

Talking about removal of certain features, I have decided to go ahead and remove the *ID_code* column from both train and test datasets as these are just the serial numbers and do not contribute anything to the classification process.

After the drop, we are left with 200000 observations and 201 (200 independent + 1 dependent) features in the train dataset, and 200000 observations and 200 (all independent) features in the test dataset.

The classes in the *target* column of the train dataset were also converted (0 to "No" and 1 to "Yes") for ease of readability.

2.3. Target Class Imbalance Problem

The check for unique values in the *target* column of the train dataset yielded some very interesting observations. The target variable, which represents whether a customer purchased a service or not, shows a huge imbalance in favor of *No*. Graphically speaking it looks like this:

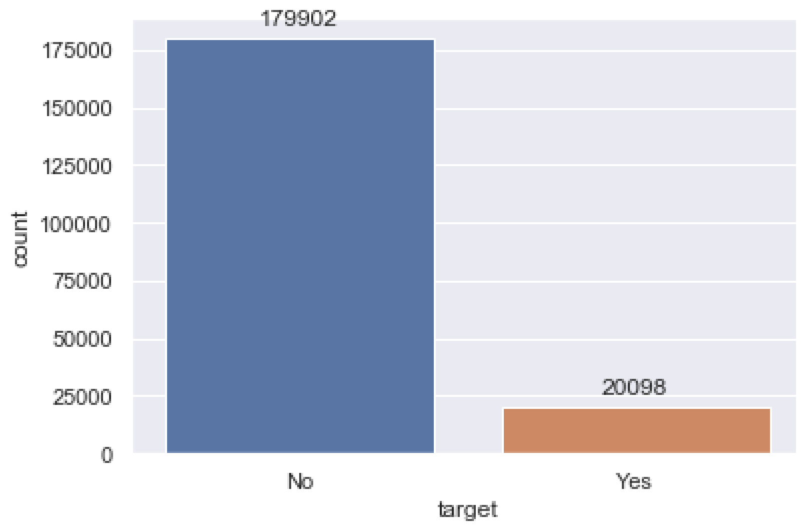


Fig 2.1. Bar Graph showing target class imbalance problem

Speaking in terms of percentage, a whopping 89.951% of the target class belongs to the majority ***No*** class while only a meagre 10.049% belongs to the minority ***Yes*** class. This is a case of target class imbalance and leads to problems while applying the model on the test data. This is because while training the model on the train data, it will be hugely biased towards the majority *No* class, and will only consider the 10% minority *Yes* class as noise and disregard it. Therefore the model will not be able to correctly predict for future test cases as it will more or less classify everything as Yes.

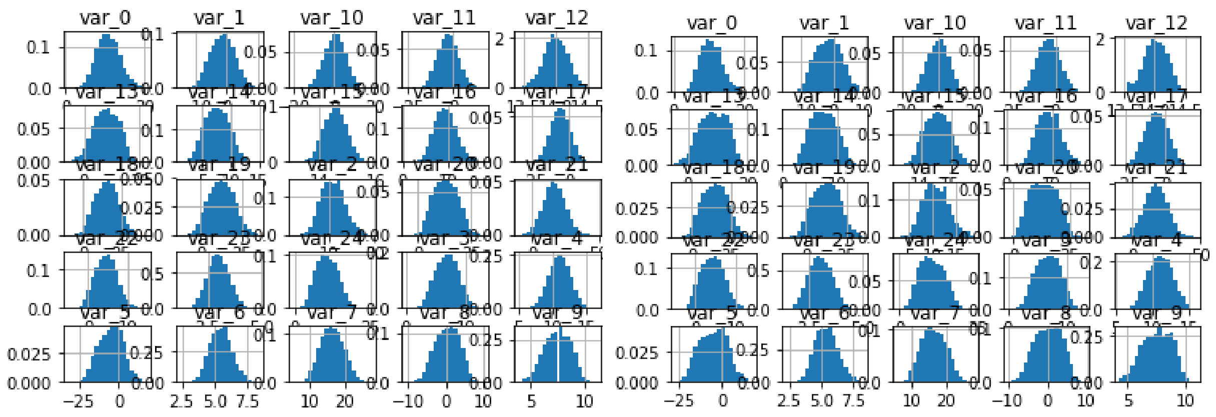
In order to fix this, **SMOTE (Synthetic Minority Oversampling Technique)** will be done on the training dataset and this modified trained dataset will be carried forward with the preprocessing and the model application. This then will be validated on the test data. (NOTE: Random Undersampling will not be done as it leads to massive loss of data. Random Oversampling will not be done either as this leads to overfitting as a lot of duplicate instances of the minority class is created.)

The principle behind SMOTE is that instead of just replicating the minority observations and bringing their count up to the count of the majority observations, it creates synthetic observations based on the existing minority observations. For each minority class observation present in the dataset, SMOTE determines the k nearest neighbors in its vicinity, connects the original point with each of these neighbors and creates a synthetic point between them. This is how synthetic instances of the minority class observations are created.

Thus after applying SMOTE on the train dataset, the number of observations increased from 200000 to 359804 (179902 from Yes class + 20098 from No class + 159804 synthetic observations of No class). As a result, now an equal number of 179902 observations each were present from the Yes and the No class.

2.4. Data Visualization

Once ***exploratory data analysis*** is done on the datasets, we take a look at the distribution of some of the variables involved. For this, we can use histograms from the seaborn library in Python. This is done to get a fair idea of the variables involved, and see if they are normally distributed or not, or skewed or not. Here are the first 25 variables for the train and test datasets.



(a) *(b)*

Fig 2.2 Histogram plot showing distrinution of first 25 variables of (a) train and (b) test datasets

After taking a look at the first 25 variables for both train and test datasets, we can extrapolate this information to all the 200 variables involved and get a general idea for all of them. We can thus come to the conclusion that all variables present in both datasets are normally distributed within themselves.

(c)

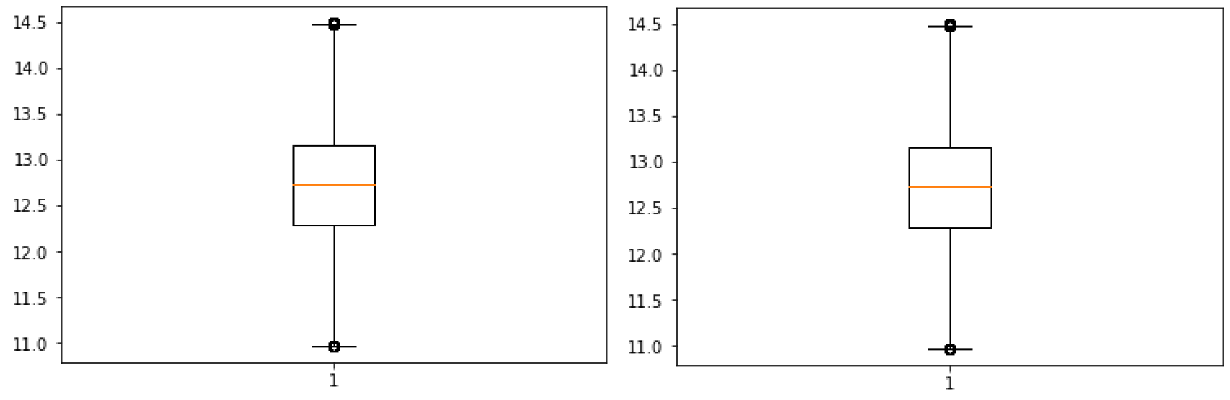
CHAPTER III

DATA PREPROCESSING

3.1. Outlier Analysis

Outliers in a dataset are nothing but data points that differ significantly from other observations. They almost always tend to cause anomalies in the results obtained after the application of Machine Learning algorithms. Outliers in a dataset may arise due to defective apparatus, data transmission errors, changes in system behavior, fraudulent behavior, human errors, etc. They may also arise due to false assumptions in the programmer or researcher's theory. In statistics, outliers can be best detected by using boxplots to plot the features involved in a dataset.

Functions were written in Python for viewing boxplots (see appendix A and B) using the Python function *matplotlib.pyplot.boxplot*. Let us see the boxplot for random features *var*_50 in the train dataset and *var*_50 in the test dataset.



(a) *(b)*

Fig 3.1 Boxplot chart showing outliers for var_50 in (a) train and (b) test datasets

From the above boxplots and by looking at boxplots for many other features present in both datasets, we come to the conclusion that all the variables have the presence of outliers in them. In order to fix this, I wrote a code (see Appendix C) which detects these outliers and replaces them with a NULL value. Next I made a known value NULL and imputed it using the median method and then the mean method. **The mean method gave a more accurate result and so I froze that method and used it to impute all previously made NULL values.**

3.2. Feature Selection

The next important thing to do before proceeding any further is feature selection. Here we assess the importance of each predictor in our dataset and remove the ones which do not contribute a whole lot to the overall prediction process. There are two important things to keep in mind:

- All independent variables should have low correlation with each other. In layman terms, if two independent variables have high correlation, that means that they convey the same meaning and in such a case, one of them can be dropped without affecting the overall prediction. **A correlation analysis test using a heatmap plot was made to check this.**
- Every independent variable should have a fairly high correlation with the dependent variable. If this happens, that means that the dependent variable can be well explained by that independent variable. So in other words, if any independent variable has a low correlation with the dependent variable, it cannot explain the dependent variable well enough, and thus it can be dropped. **A feature importance test was done to check this.**

3.2.1. Correlation Analysis

A correlation analysis test using a heatmap plot was used here to check the dependence amongst the independent features. A function was created (see Appendix D) which would take two row indices of the train dataset as parameters and give out a heatmap showing the correlation amongst all variables between those two row indices.

For example, if the indices 1 and 6 are passed to the function, the heatmap would have correlation for **var_0** (index = 1), **var_1** (index = 2), **var_2** (index = 3), **var_3** (index = 4) and **var_4** (index = 5). So any chunk could be passed and the correlation could be seen. This was done because it is impossible to view a correlation plot for all 200 variables involved.

The following is a heatmap showing the correlation amongst variables **var_0, var_1, var_2, var_3** and **var_4**.

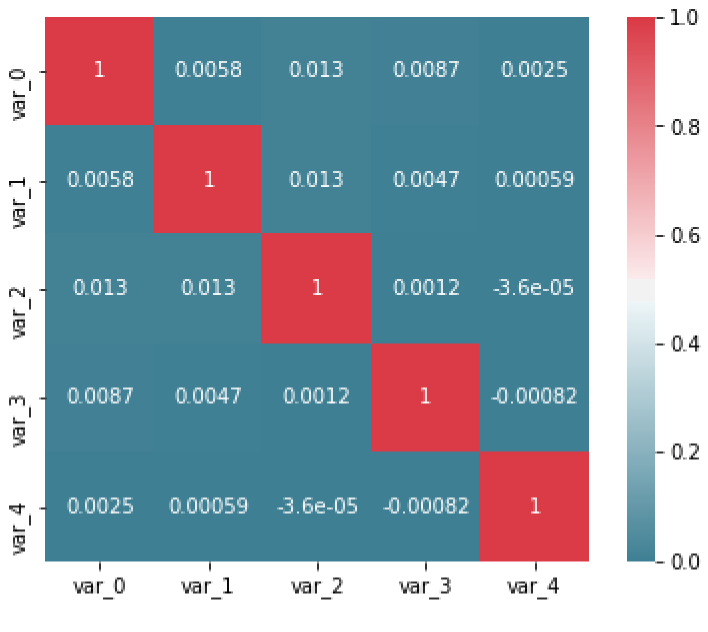


Fig 3.2 Heatmap showing correlation for first 5 variables in train dataset

After multiple iterations where different chunks of the independent variables were passed, I came to the conclusion that all independent variables have very less correlation with each other. For example, the correlation plot given above clearly shows that the correlation coefficients are very very low. So no two variables are similar to each other and thus all variables can be kept as of now.

3.2.2. Feature Importance

The next step was to check for the importance of each independent variable in predicting the target class. **ExtraTreesClassifier** is used from the inbuilt feature importance class in Tree Based Classifiers (see Appendix E). This gives a score for each feature of the data, with the highest score being for the most important feature.

Since 200 variables is a lot for the model to analyze, taking advantage of the importance scores here, I will solve the rest of the way going forward in two different ways.

- Top 60% variables (120 variables) will be taken moving forward and the rest will be dropped.
    - This would give me 359804 observations and the top 120 columns based on highest importance scores. A subset of this was stored in the dataframe *new_train_1*.
- A threshold value of the mean of importance scores will be taken and anything below the mean will be dropped.
    - This approach left me with 359804 observations and the top 74 variables based on highest importance scores. A subset of this was stored in the dataframe *new_train_2*.

After this, the same features were also subsetted out from the test dataset. Thus we now have two pairs of datasets. One model would be trained on *new_train_1* and then applied on *test_1* (with

359804 observations and 120 features). A second model would be applied on *new_train_2* and then applied on *test_2* (with 359804 observations and 74 features).

3.3. Feature Scaling

The next step in data preprocessing is known as Feature Scaling. Typically, this is done when we have continuous independent variables having different ranges. For example, let us say we have an *age* variable in our dataset, ranging from 0-99 and an *income* variable which ranges from 10000 – 400000. This biases the output heavily towards *income* because it has higher ranges, as compared to *age*. In such a scenario we scale both the variables to have an equal range.

In our case, all variables in both pairs of datasets have varying ranges of mean, standard deviations, minimum, maximum, etc. Thus it is very important to scale these values to a common range. Either normalization or standardization needs to be done. Taking a look at the histograms for the variables in Fig 2.2, all the variables seem to be normally distributed. Therefore, **standardization** is employed to bring all variables to a uniform range. **Standardization** is done so as to make all variables have a standard normal distribution, with mean = 0 and standard deviation = 1. This was done on both pairs of datasets present in our case.

3.4. Train/Test Splitting

This is the final step before applying Machine Learning algorithms on our dataset. The train datasets (*new_train_1* and *new_train_2*) need to be split in an 80-20 way. The reason for this is simple. The two respective test datasets have no target variable in them and there is no way the credibility or accuracy of the models can be determined. So 80% of the train datasets would be used to train the models, and then the models would be implemented on the other 20% so as to determine their accuracies. Then and only then will the models be applied on the test datasets.

The approach going forward with the implementation of machine learning algorithms would be to train our first model on 80% of *new_train_1*, then apply it on the remaining 20% of *new_train_1* so as to determine the accuracy metrics like AUC Score, Precision and Recall. This model would then be applied on *test_1*.

Similarly, a second model would be trained on 80% of *new_train_2* and then applied on its remaining 20% to determine the accuracy metrics. Then, it would be applied on *test_2* to predict the target labels.

Here, I have used *train_test_split* from *sklearn.model_selection* and used *simple random sampling* to do the splitting.

CHAPTER IV

APPLICATION OF MACHINE LEARNING ALGORITHMS

4.1 Introduction

Problems in data science can be broadly classified into two types – classification and regression. This project predicts whether a customer would make a transaction or not, irrespective of the amount of money transacted. The dependent variable *target* has two classes, “Yes” or “No”. So this problem statement falls under the umbrella of a binary classification problem. The following Machine Learning algorithms were applied on both pairs of datasets:

- Logistic Regression
- Decision Tree Classification
- Naïve Bayes Classification

Then, the AUC Score, Precision and Recall were calculated.

Let us consider each of the above algorithms and see the accuracies involved.

4.2. Logistic Regression

The logistic regression model is a supervised machine learning algorithm which inspects the relationship between linearly weighted independent variables and an output variable. The output here is modeled as a binary value (0 or 1), as compared to a linear regression model where the output is numeric in nature. The logistic function used for logistic regression is an S-shaped curve, much like a population growth curve, which rises exponentially and then maxes out at the carrying capacity of the environment. In our case of multiple independent variables and one dependent variable, the regression equation behind this would look like:

$$p(x) = \frac{\exp(b_0 + b_1X_1 + b_2X_2 + \cdots + b_nX_n)}{1 + \exp(b_0 + b_1X_1 + b_2X_2 + \cdots + b_nX_n)}$$

where, p(x) is the probability of the predicted output, x_1, x_2,..., x_n are the dependent variables, b_1, b_2..., b_n are their respective slopes, and b_0 is the y-intercept. Each slope represents the effect of the respective variable on the dependent variable, in other words, if x_1 increases (or decreases), y increases (or decreases) proportionally by b_1 units. And if all the dependent variables are 0, then y is equal to the intercept, b_0.

Using this equation, we build a model on the training data and try to find probability of each class in the target column as a function of the independent variables. Here, 0.5 was set as the threshold probability, so anything with probability between 0 and 0.5 was taken as 0 (or No) and anything between 0.5 and 1 was taken as 1 (or Yes).

The *statsmodels.api* module was imported for this. The code can be found in Appendix F. The following metrics were obtained:

	Model 1	**Model 2**
AUC Score	0.784	0.761
Precision	0.773	0.755
Recall	0.798	0.773

Table 4.1. Table showing metrics for Logistic Regression models

4.3. Decision Tree Classification

Decision tree-algorithms also fall under the category of supervised machine learning algorithms. Using the CART algorithm (Classification and Regression trees), they can be used for both classification and regression models. It calculates the entropy for every predictor variable at each stage and then splits the predictor which has the maximum information gain. In theory, this holds true because the predictor variable that gives out the most information is chosen for splitting. This process continues until no further splits can be done.

The *sklearn.tree* module was imported and *DecisionTreeClassifier* was imported from there. Two decision tree models were made, one for the data with 120 variables and other for the data with 74 variables (see Appendix G). Once the model was used to learn on 80% of the train data, it was applied on the remaining 20%. Accuracy metrics were obtained. Then the model was applied on the test data.

The following are the metrics obtained:

	Model 1	**Model 2**
AUC Score	0.773	0.770
Precision	0.748	0.751
Recall	0.814	0.811

Table 4.2. Table showing metrics for Decision Tree models

4.4. Naïve Bayes Classification

Naïve Bayes classifiers are probabilistic machine learning algorithms that are based on the Bayes theorem.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

where, P(A|B) is the probability of A happening, given that B has occurred.

The advantage of using Naïve Bayes Classifiers is that they are fast and easy to implement but their limitation is that the predictors need to be independent. This is not a problem in our case, as the correlation analysis done earlier showed that all predictors were indeed independent.

Since we have multiple independent variables here, the probability for let's say class ***No*** in the first dataset would be:

$$P(No|var\_0, var\_1, \dots, var\_199) = \frac{P(var\_0|No)P(var\_1|No), \dots, P(var\_199|No)P(No)}{P(var\_0)P(var\_1), \dots, P(var\_199)}$$

This above equation is used to calculate the probabilities for each event for each class and the class with the higher probability is chosen for the respective test values. The *sklearn.naive_bayes* module was imported and *GaussianNB* was imported from there (see Appendix H). The following metrics were obtained:

	Model 1	**Model 2**
AUC Score	0.848	0.822
Precision	0.898	0.846
Recall	0.783	0.788

Table 4.3. Table showing metrics for Naïve Bayes Classifier models

4.5. Choosing the best model

To summarize, here are the metrics for all three algorithms on both pairs of datasets:

	Logistic Regression	
	Model 1	Model 2
AUC	0.784	0.761
Precision	0.773	0.755
Recall	0.798	0.773
	Decision Tree	
	Model 1	Model 2
AUC	0.773	0.77
Precision	0.748	0.751
Recall	0.814	0.811
	Naïve Bayes	
	Model 1	Model 2
AUC	0.848	0.822
Precision	0.898	0.846
Recall	0.783	0.788

Table 4.4. Table summarizing evaluation metrics for all models used

Let us define each accuracy metric before making a pick of the best model.

1. AUC Score

One of the most essential performance metrics of a classification model is its AUC score. To begin understanding the concept behind the AUC score, let us first understand what an ROC curve is.

Suppose we have a model which predicts whether a certain food item is edible or not. Plotting a probability distribution curves and their ROC curves, we get the following:

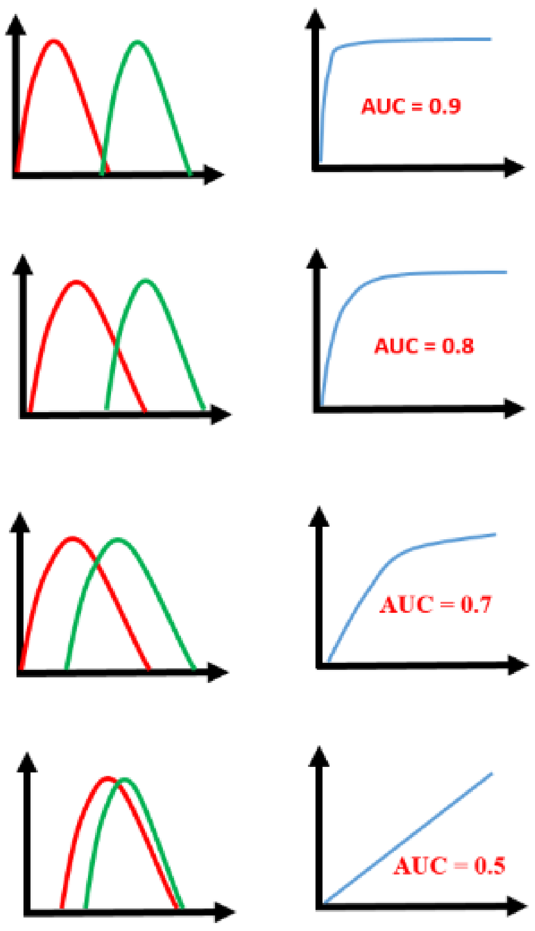


Fig 4.1 Probability distribution curve v/s ROC curve

On the left hand sides, we have the probability distribution curves, where the red curve represents the food item being not edible, and the green curve represents the food item being edible. The right hand curves are made between ***Sensitivity*** or ***True Positive Rates*** (which is the proportion of food items that were predicted correctly to be edible upon food items that are actually edible) on the y-axis v/s ***False Positive Rates*** (the proportion of food items that were falsely classified as edible, but aren't, upon food items that are actually not edible) on the x-axis. The AUC score is the area under the ROC curve. It lies between 0 and 1.

The job of any model is to correctly distinguish between Yes and No, which in this case, are the probabilities of the food item being edible or not. In the first case, the model does a very good job of distinguishing between red and green (or between not edible and edible), so it's AUC score is 0.9. As we go down, the overlap between the red and the green curves increase, and as such the model cannot really do a good job of distinguishing between the two classes. Therefore, the AUC score also decreases.

2. Precision

Going back to the previous example of whether the food item is edible or not, let us take a look at the following figure:

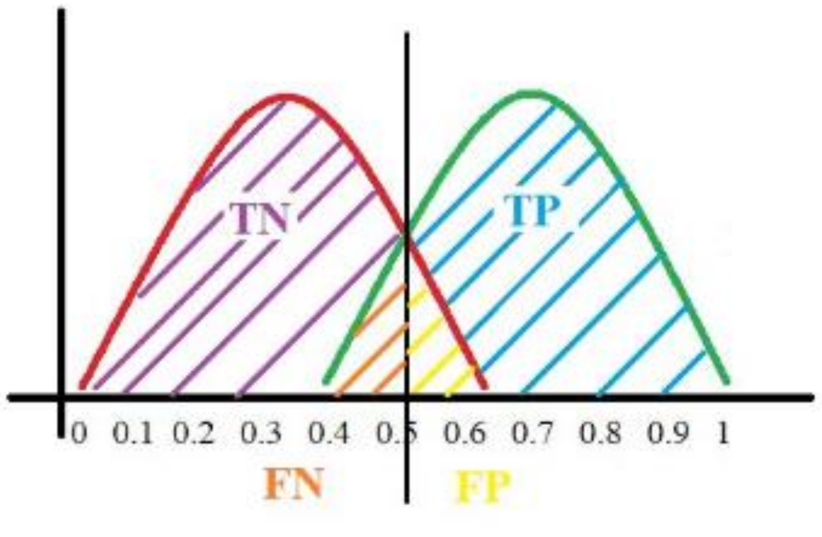


Fig 4.2. Probability distribution curve showing all four regions of TP, TN, FN, FP

Once again, the red curve represents non-edible food items, and the green curve represents edible food items. Setting a threshold line at exactly the middle, the model predicts everything to its right as edible, and everything to the left of it as non-edible. Taking a look at the marked regions:

- TP (True Positives) are cases which are actually positive (edible) and have been predicted correctly.
- FP (False Positives) are cases which are actually negative (non-edible) but have been predicted falsely as positives.

- TN (True Negatives) are cases which are actually negative (non-edible) and have been predicted correctly.
- FN (False Negatives) are cases which are actually positive (edible) but have been predicted falsely as negatives.

Now, precision is used to answer what proportion of positively identified cases were actually positive. In other words, out of all the cases where the model predicted the outcome as edible food, what percentage of it was actually edible? Mathematically speaking,

$$Precision = \frac{TP}{TP + FP}$$

3. Recall

Recall lays out what proportion of actually positive cases were identified correctly as positive. In other words, out of all cases where the food item is actually edible, what percentage of it was predicted as edible? Mathematically speaking:

$$Recall = \frac{TP}{TP + FN}$$

The best model

The bottomline is, all three of these metrics are very important when determining the success of a model. A lot of the times, everything depends on the requirement of the business. Sometimes clients are just concerned with False Negatives. A good example of this is the Santander Customer Transaction Prediction example itself, where the model determines whether a customer would make a transaction in the future or not. A high False Negative means that the model predicted a lot of customers as not making a transaction, whereas in reality they would, and thus this results in massive losses, as these were people who would allegedly buy a future product. Similarly at other times, businesses look for other accuracy metrics like Precision, Recall, Accuracy, etc. It should be noted that no one accuracy metric can be a hard-and-fast helper in these cases, as a combination of all metrics is required to determine the best performing model.

All-in-all, out of all three models that were used here on both pairs of datasets, the ***Naïve Bayes Classifier*** on the first pair of datasets gave the best results. This is the pair of datasets which had 120 variables in it (the rest 80 were disregarded after doing a feature importance test). For this case, all three metrics used gave the highest scores, (AUC Score: 0.848, Precision: 0.898, Recall: 0.783), and as such this would be considered as the best performing model.

CHAPTER V

CONCLUSION

Initially a train dataset having 200000 observations and 201 features (200 independent + 1 dependent) and a test dataset having 200000 observations and 200 independent features were given for this problem. After removing outliers from both datasets and imputing them with their mean, a feature importance test was done which would give out the scores for all 200 variables involved, in terms of their importance in determining the target class. Two separate cases were taken, one having the top 120 variables, and the other having top 74 variables (the mean of importance scores was taken and anything below that was dropped). After standardizing the variables to rescale and center them, both train datasets were split in an 80-20 way. Three different models were applied on 80% of the data, then validated on the remaining 20%. Once the performance metrics were determined, these models were applied on the respective test datasets. Out of all these, the Naïve Bayes Classifier model on the dataset with 120 variables was chosen as the best model.

While performance metric values lying between 0.8 and 0.9 seem to suggest that the application of the model was good, there is a lot of room for improvement. While some variables were removed here for ease of computation and time taken, few other models could be made by considering all 200 variables involved. Even some parameters can be tuned in the existing models to see if better results are acquired. Also, some other algorithms could be tried like KNN Classifier, XGBoost, LightGBM, etc. These are but only a few suggestions and there are indeed possibilities for a lot more.

All in all, understanding how the data behaves and making appropriate changes as we go along is how proper analyzing should be done, and therefore changes could be made here and there in bits and pieces to improve the overall model accuracy.

References

- McKinney, W. (2012). *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. "O'Reilly Media, Inc.".
- Grolemund, G., & Wickham, H. (2017). *R for data science: import, tidy, transform, visualize, and model data*. "O'Reilly Media, Inc.".

Appendix A

Function to check outlier on train data

```
#function to check outlier
def outl(var_name):
    %matplotlib inline
    plt.boxplot(new_train[var_name])
    plt.show()

#calling the function
outl("var_50")
```

Appendix B

Function to check outlier on test data

```
#function to check outlier
def outly(var_name):
    %matplotlib inline
    plt.boxplot(test[var_name])
    plt.show()

#calling the function
outly("var_50")
```

Appendix C

Code to make outliers NaN and display their count

```
#loop to make outliers NaN
dic = {}
count = 0
c = 1
k = new_train.columns[1:]
# print(k)
for i in k:
    print(i)
    q75, q25 = np.percentile(new_train[i], [75, 25])
    print("q75 = ", q75, "q25 = ", q25)
    iqr = q75 - q25
    print("iqr = ",iqr)
    mini = q25 - (iqr * 1.5)
    maxi = q75 + (iqr * 1.5)
    print("minimum = ",mini, "maximum = ",maxi)
    for j in range(0, len(new_train)):
```

```python
        if new_train.iloc[j, c] > maxi or new_train.iloc[j, c] < mini:
                   new_train.iloc[j, c] = np.nan
                   count += 1
    print("Count of", i, "=", count, "\n")
    dic[i] = count
    count = 0
    c += 1
print(dic)
```

Appendix D

Function to display correlation plot using heatmap

```python
#function to build heatmap
def corr_plot(index1, index2):
    col = new_train.columns[index1 : index2]
    sub_col = new_train.loc[0:359804, col]

    ##building correlation plot
    #setting width and height of plot
    f, ax = plt.subplots(figsize = (7, 5))

    #generate the corr matrix
    corr = sub_col.corr()

    #putting corr into perspectivev via heatmap
    sns.heatmap(corr, mask = np.zeros_like(corr, dtype = np.bool), cmap = sns.diverging_palette(220, 10, as_cmap = True), square = True, ax = ax, annot = True, cbar = True)

#calling
corr_plot(1, 6)
```

Appendix E

Code to print out feature importance scores

```python
#splitting new_train into ndependent and dependent variables
X = new_train.iloc[:,1:201]  #independent columns
y = new_train.iloc[:,0]    #target column

#importing required function
from sklearn.ensemble import ExtraTreesClassifier

#building model
model = ExtraTreesClassifier()
model.fit(X,y)

#printing out the importance scores
print(model.feature_importances_)

#creating series for better visualization
feat_importances = pd.Series(model.feature_importances_, index=X.columns)
```

```python

#plotting graph of feature importances for better visualization
feat_importances.nlargest(10).plot(kind='barh')    #top 10 features
plt.show()

#printing feat_importances to get another idea
print(feat_importances)

#sorting
feat_imp_sorted = feat_importances.sort_values(ascending = False)
print(feat_imp_sorted)
```

Appendix F

Code for building Logistic Regression model

```python
#importing required library
import statsmodels.api as sm

#building 1st logistic model
logit_model_1 = sm.Logit(nt1_train.iloc[:, 120], nt1_train.iloc[:, 0:120]).fit()

#checking summary
logit_model_1.summary()

#applying model on nt1_test
nt1_test["actualprob"] = logit_model_1.predict(nt1_test.iloc[:, 0:120])

#convert probs to 0 and 1
nt1_test["predval"] = 1
nt1_test.loc[nt1_test.actualprob < 0.5, "predval"] = 0

#applying model on test
test_1["actualprob"] = logit_model_1.predict(test_1.iloc[:, :])

#convert probs to 0 and 1
test_1["predval"] = 1
test_1.loc[test_1.actualprob < 0.5, "predval"] = 0
```

Appendix G

Code for building Decision Tree model

```python
#importing from library
from sklearn.tree import DecisionTreeClassifier

#applying model on nt1_train
DT_model_1= DecisionTreeClassifier().fit(nt1_train.iloc[:, 0:120], nt1_train.iloc[:, 12
0])

#checking summary
```

```
DT_model_1

#applying model on nt1_test
DT_predict_1 = DT_model_1.predict(nt1_test.iloc[:, 0:120])

#applying model on test
DT_predict_on_test_1 = DT_model_1.predict(test_1.iloc[:, :])
```

Appendix H

Code for building Naïve Bayes Classifier model

```
#importing required library
from sklearn.naive_bayes import GaussianNB

#building 1st NB Model
NB_model_1 = GaussianNB().fit(nt1_train.iloc[:, 0:120], nt1_train.iloc[:, 120])

#summary
NB_model_1

#predicting on nt1_test
NB_predict_1 = NB_model_1.predict(nt1_test.iloc[:, 0:120])

#predicting on test
NB_predict_on_test_1 = NB_model_1.predict(test_1.iloc[:, :])
```