Machine Learning Engineer Nanodegree

Capstone Project

Adriel Vieira

August 15th, 2018

I. Definition

Project Overview

This project is based on a Kaggle competition posted by Santander bank. The goal of this competition is to help Santander predict whether a given customer is a satisfied or unsatisfied customer. Customer satisfaction is an important measure for businesses. Unhappy customers are easy to lose and rarely voice their dissatisfaction before leaving.

By helping to identify dissatisfied customers early in their relationship, we give Santander the opportunity to take proactive steps and improve a customer's happiness before it's too late. Reducing churn is something really important for most companies, considering that acquiring new clients is much more expensive than working on the current relationships and keeping them active and profitable.

Problem Statement

In this competition, the task is to work with hundreds of anonymized features to predict if a customer is satisfied or dissatisfied with their banking experience. It is a binary classification problem in which the goal is to predict the probability that each customer in the test set is an unsatisfied customer.

Since we have hundreds of features to work with, and we don’t have information about the meaning behind these features, the steps of feature selection and feature engineering will be really important for this task.

The datasets are available at www.kaggle.com/c/santander-customer-satisfaction/data.

The solution to this machine learning problem is a binary classifier that predicts a value of 1 or 0 (and/or the probabilities of these labels being true), representing unsatisfied and satisfied customers. After working on features, I'll try different algorithms (e.g. linear regression, random forest and XGBoost), with different sets of parameters. The combination that performs better will be used to predict on the testing set, generating our submission to Kaggle.

Metrics

This competition's submissions are evaluated on area under the ROC curve between the predicted probability and the observed target.

The model estimates a probability of the target variable being True (or 1) and a threshold value is then applied to separate Trues and Falses. The ROC curve is the interpolated curve made of points whose coordinates are functions of the threshold. It is obtained by plotting the True Positive Rates (True Positives / All Positives) on the Y axis and False Positive Rates (False Positives / All Negative) for every probability threshold value possible to be used to classify a datapoint as Positive or Negative.

The usage of AUC as scoring method relates to the nature of the problem. It gives Santander the autonomy to choose which threshold to use to classify a customer as happy or unhappy. Both a large and a small threshold value are risky: the bank could lose clients if the value is too high, or spend too much money to tackle lots of customers that are already satisfied. A model trained on ROC AUC is a model optimized for any threshold the bank chooses to use.

II. Analysis

Data Exploration

Santander provided an anonymized dataset containing a large number of numeric variables (370 features and 1 target variable). The "TARGET" column is the variable to predict. It equals one for unsatisfied customers and 0 for satisfied customers. The dataset provided has 76020 rows or data points for training. The testing set provided contains roughly as much data points as the training set (75818 rows), but lacks of the "TARGET" column. This dataset will be used as validation set.

The number of rows represents the number of customers in the dataset. We will train our algorithms on the data of more than 76000 customers, of which 96% are classified

as satisfied. It’s a highly unbalanced dataset and this should be taken into account while training our models.

Each customer has a unique ID, which has been dropped from the training set. The remaining features are mostly integers (260) and the rest is represented by floats.

More than a hundred integer features are binary and there’s little to do on preprocessing them. Some features (34) have only one value, and so are useless and therefore have been dropped. Many others (74 features) have between 3 and 9 unique values. I have treated these as categorical and further applied One Hot Encoding to them.

Among floats, more than 90% of the data is 0, while the median of the maximums of each feature is around 400,000. Because of this unbalancing behavior that also appears on features it is hard to decide to drop any outlier, since we could be dropping useful and rare information for a problem that seems like an anomaly detection challenge. Some values are pretty weird, though. Numbers like -999999 and 9999999999.00 shows up on features that don’t have anything close to that. These values have been treated as an anomaly of the data and been replaced by the mode of each feature.

The dataset has no missing values, but have duplicated columns which have been removed.

Exploratory Visualization

The dataset provided has lots of features, and although these are anonymized variables, it seems that features' names still carry some meaning. I noticed that all features' names have the word "var" followed by a number. To check if features related to the same "var" have some relationship, I analyzed the correlation between all the features in the dataset, using pd.DataFrame.corr(), and then I grouped and summed this data frame according to the number followed by the word "var" in its index. The result of this a data frame showing the sum of correlations between a given feature and all the features that have a given number following the word "var" in its name. For example:

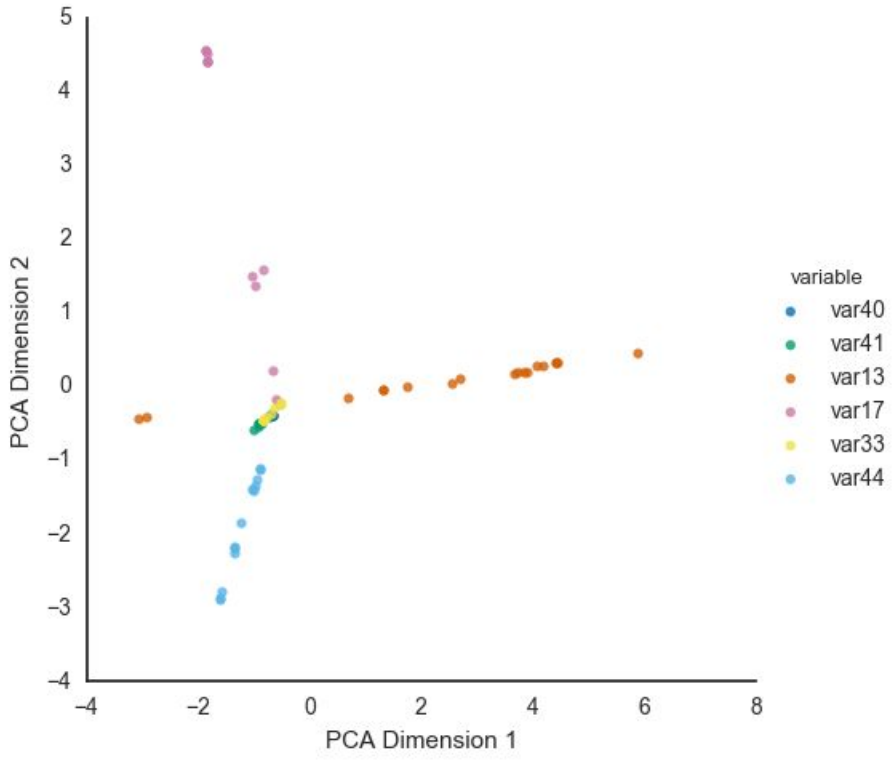
variable	var13	var17	var33	var40	var41	var44
imp_op_var40_comer_ult1	0.086057	0.033690	-0.006795	3.493097	0.414791	-0.003133
imp_op_var40_comer_ult3	0.072513	0.011480	-0.007530	3.346345	0.342237	-0.004127
imp_op_var40_efect_ult1	-0.013730	-0.000691	-0.000796	2.626585	0.259757	-0.001587
imp_op_var40_efect_ult3	-0.015849	-0.000796	-0.000917	2.792837	0.315941	-0.001827
imp_op_var40_ult1	0.093516	0.032167	-0.000852	3.766482	0.370634	-0.002836

In this case, I'm using only var13, var17, var33, var40, var41 and var44, which are the ones that have more features related to it.

To visualize it in a single plot, I transformed these 6 columns into 2 dimensions, using PCA.

	PCA Dimension 1	PCA Dimension 2
imp_op_var40_comer_ult1	-0.672445	-0.381212
imp_op_var40_comer_ult3	-0.666660	-0.389659
imp_op_var40_efect_ult1	-0.703021	-0.371338
imp_op_var40_efect_ult3	-0.717669	-0.382699
imp_op_var40_ult1	-0.676120	-0.392336

I plotted the resulting data frame above into a scatter plot, in which the hue was set to which “var” number was present in the index.



The resulting visualization shows that the number followed by the word “var” in features’ names can be used to cluster these features. I used this insight to reduce the number of features without losing too much information by applying PCA to each group of features, grouped by the the number following “var” in its name.

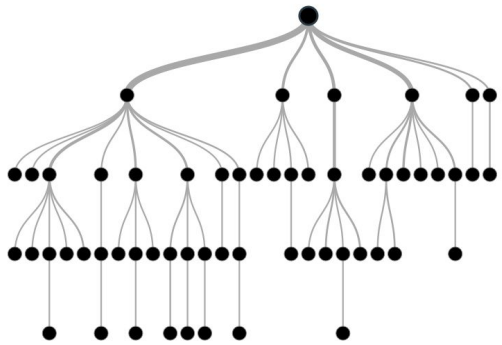
Algorithms and Techniques

In order to solve the classification task, several classifiers were be tested to check which one performs better.

A Logistic Regression algorithm is easy to understand and will be tried out. It takes features and adjusts coefficients and an intercept to estimate an outcome. The problem with Logistic Regression is that it requires a linear separable problem to perform well.

To deal with more complexes data analysis I used two different tree based models: Random Forest and XGBoost. I have used scikit learn implementation of K Fold Validation to validate these models.

Tree based models can handle classification and regression problems and are extensively used in machine learning. They are easy to understand and its graphical representation makes it greatly and intuitively understood.



Decision trees can handle both numerical and categorical input variables and doesn't require lots of data cleaning as other branches of algorithms, like neural networks for example. Also, they are called non parametric methods, which means that no assumption is made about the data distribution.

A decision tree is made of recursive data splits that try to reduce the impurity of each data sub-sample. After a number of splits, the expectation is to have buckets of data in which most samples belong to one target group. In our case, the groups are Satisfied (0) and Unsatisfied (1) customers.

Decision trees split data based on categorical and continuous features. For continuous features, the algorithm will define discrete set of intervals to partition the data. Good splits will lead to homogeneous samples in respect to the target variable. To measure the quality of each split, a measure of impurity is needed - the lower the impurity, the better the split. The impurity measure depends on the taks being solved (classification or regression) and in our case, the Gini Index will be used. The Gini index measures how likely it is that two randomly chosen items belong to the same class.

Decision trees are the building blocks for other methods called Ensemble Methods. This basically means combining different models to achieve a better and more reliable prediction. Bagging (Random Forest) and Boosting are commonly used ensemble methods and were used for this project.

In Random Forest, as it name suggests, several Decision Trees are trained. Each tree is trained on a random subset of the data and its predictions are used as votes for classifications.

Assuming N cases in the training set, each tree receives as input a random sample of N, taken with replacement. The trees are trained only on a random portion of features. The forest will classify each sample with the label that receive most “votes” from trees.

Random Forests grows several Decision Trees in parallel, resampling the data over and over. It creates several classifiers with high variance and low bias that are prone to overfitting. The main idea is that these classifiers overfit in different ways and after voting, these differences are averaged out.

In the other hand, boosting is sequential, not parallel like Random Forest, and refers to a family of algorithms that use weak learners together in order to build a strong learner.

The idea behind boosting is to train a base weak learner (with high bias and low variance), and sequentially train other weak learners on the residual of the previous one. Each classifier is trained at a time, with the goal of improving the previous predictions.

Benchmark

Kaggle provides a Sample Submission to be used as benchmark for the project. In our case, the benchmark is an all-zeros submission. Our training set has much more ones than zeros and it also makes sense to think of an all-zeros scenario as the banking taking no action on customers’ behavior and considering them all “satisfied” customers.

III. Methodology

Data Preprocessing

Cleaning

As mentioned before, the data provided has some weird values in it that were replaced by the mode of each feature, using the following function:

```
def replace_weird(dfs, weird = [-999999, 9999999999.00]):
    for df in dfs:
        modes = df.mode()
        for col in df.columns:
            if any([i in df[col].values for i in weird]):
                df[col].mask(df[col].isin(weird), modes[col][0], inplace=True)

replace_weird([train, test])
```

Columns with the same value for every datapoint won't help an algorithm to learn, and neither will a duplicated feature. These columns have been removed using the following functions and scripts:

```
def remove_useless(dfs):
    remove = []
    for df in dfs:
        for col in df.columns:
            if df[col].std() == 0:
                remove.append(col)
    for df in dfs:
        df.drop(set(remove), axis=1, inplace=True)

remove_useless([train, test])
```

```
def duplicated(df):
    duplicated = []
    c = df.columns
    for i in range(len(c)-1):
        v = df[c[i]].values
        for j in range(i+1,len(c)):
            if np.array_equal(v,df[c[j]].values):
                duplicated.append(c[j])
```

```
    return set(duplicated)
duplicated_train = duplicated(train)
duplicated_test = duplicated(test)

if duplicated_test.issubset(duplicated_train):
    print('Duplicated columns on test set are also duplicated on training set.\
          \nAll duplicated being removed from both sets')
    train.drop(duplicated_train, axis=1, inplace=True)
    test.drop(duplicated_train, axis=1, inplace=True)
```

I have noticed that some integer features seemed to be a function of float features related to the same “var” group. I decided to build a regressor to try to predict each integer of the dataset using the floats as inputs. I used this method to get rid of unnecessary variables that could be easily predicted using the rest of the data. I have used the code below to do so:

```
from sklearn.cross_validation import train_test_split
from sklearn.tree import DecisionTreeRegressor

useful_integers = []

for c in train.select_dtypes('int').columns:
    X_train, X_test, y_train, y_test = train_test_split(train.select_dtypes('float'),
                                                        train[c],
                                                        test_size = 0.25,
                                                        random_state=3)
    regressor = DecisionTreeRegressor(random_state=3)
    regressor.fit(X_train, y_train)
    score = regressor.score(X_test, y_test)
    if score<0.9: useful_integers.append(c)
```

Extra Features

Some extras features have been calculated, based on operations on each row.

Since we’re dealing with a bank problem, it is expected that a relevant portion of the data might be representing amounts of money, so the feature values_sum was created in order to represent the sum of these hypothetical amounts across features plus the noise caused by data that doesn’t represent money.

```
def values_sum(df):
    df['values_sum'] = df[features].sum(axis=1).astype(float)
```

Banks usually have lots of different financial products and the amount of products a given customer uses might impact its satisfaction probability. I created a feature named “zeros” that is

the sum of features with value equals to zero in a given row. Hypothetically, this feature would give information on how little involved to the bank one client is.

```
def count_zeros(df):
    df['zeros'] = (df[features]==0).sum(axis=1).astype(float)
```

Another attempt to combine currency related features was to use features with the word “saldo” in it. Not only the word “saldo” has been used but also the terms “hace2”, “hace3”, “ult1” and “ult3”, that often came together in “saldo” features. My guess was that these words might reveal an amount of money in a specific time.

```
saldo_var_columns = [c for c in train.columns if 'saldo_var']
saldo_medio_hace2_columns = [c for c in train.columns if 'saldo_medio' in c and 'hace2' in
c]
saldo_medio_hace3_columns = [c for c in train.columns if 'saldo_medio' in c and 'hace3' in
c]
saldo_medio_ult1_columns = [c for c in train.columns if 'saldo_medio' in c and 'ult1' in c]
saldo_medio_ult3_columns = [c for c in train.columns if 'saldo_medio' in c and 'ult3' in c]

def sum_saldos(df):
    df['sum_of_saldos'] = df[saldo_var_columns].sum(axis=1)
    df['sum_of_saldo_medio_hace2'] = df[saldo_medio_hace2_columns].sum(axis=1)
    df['sum_of_saldo_medio_hace3'] = df[saldo_medio_hace3_columns].sum(axis=1)
    df['sum_of_saldo_medio_ult1'] = df[saldo_medio_ult1_columns].sum(axis=1)
    df['sum_of_saldo_medio_ult3'] = df[saldo_medio_ult3_columns].sum(axis=1)

sum_saldos(train)
sum_saldos(test)
```

Dimensionality Reduction

As mentioned before, I have used columns’ names to group features and then apply PCA to these groups in order to reduce the dimensionality of the data. I’ve written the following functions to generate as few dimensions as possible, while explaining at least 95% of the variance in the data.

```
def make_pca(df, var, var_cols, n):
    pca = PCA(n)
    pca_data = pca.fit_transform(df[var_cols])
    pca_cols =[var+'_pca_'+str(i+1) for i in range(n)]
    return (pd.DataFrame(pca_data, columns=pca_cols),
           pca.explained_variance_ratio_.sum())

def make_pcas(floats_only=True):
    train_pca = pd.DataFrame()
```

```
    test_pca = pd.DataFrame()

    df = pd.concat([train, test])
    if floats_only: df = df.select_dtypes('float')

    for var in variables:
        var_cols = [c for c in df.columns if var in c.split('_')]
        if not var_cols: continue
        n, e = (1,0)
        while e<0.95:
            _, e = make_pca(df, var, var_cols, n)
            train_pca_temp, _ = make_pca(train, var, var_cols, n)
            test_pca_temp, _ = make_pca(test, var, var_cols, n)
            n+=1
        train_pca = pd.concat([train_pca, train_pca_temp], axis=1)
        test_pca = pd.concat([test_pca, test_pca_temp], axis=1)
    return train_pca, test_pca

train_pca, test_pca = make_pcas()
```

Scaling

The data is impacted by the presence of really large values, that have not been removed because they might contain useful information for an anomaly detection kind of problem. To reduce the impact of the skewed distribution caused by these values, non-categorical data has been scaled by the log function. But first, I have scaled the pca data using MinMaxScaler and replaced zeros to one half of the absolut minimum value of the dataset, so that we have a non-negative and non-zero dataset before applying np.log.

I have used sklearn's implementation of MinMaxScaler for most features but to scale features related to the word "saldo" I have implemented the scaler by myself, so that I could scale all these features using the same base values. The code is the following:

```
# Scaling pca

min_max_scaler = MinMaxScaler()
min_max_scaler.fit(pad.concat([train, test]).loc[:,pca_columns])
train[pca_columns] = pd.DataFrame(min_max_scaler.transform(train[pca_columns]),
columns=pca_columns)
test[pca_columns] = pd.DataFrame(min_max_scaler.transform(test[pca_columns]),
columns=pca_columns)

minimum_pca = min([train[pca_columns].replace(0,1).min().min()/2,
test[pca_columns].replace(0,1).min().min()/2])
train[pca_columns] = train[pca_columns].replace(0,minimum_pca)
test[pca_columns] = test[pca_columns].replace(0,minimum_pca)
```

```
train[pca_columns] = train[pca_columns].apply(np.log)
test[pca_columns] = test[pca_columns].apply(np.log)

# Scaling extra columns
train['values_sum'] = train['values_sum'].apply(np.log)
test['values_sum'] = test['values_sum'].apply(np.log)

# Scaling saldos
saldos = ['sum_of_saldos', 'sum_of_saldo_medio_hace2', 'sum_of_saldo_medio_hace3',
'sum_of_saldo_medio_ult1', 'sum_of_saldo_medio_ult3']
min_saldo = min(train[saldos].min().min(), test[saldos].min().min())
max_saldo = max([train[saldos].max().max(), test[saldos].max().max()])
saldo_range = max_saldo - min_saldo

train[saldos] = train[saldos].apply(lambda x: (x - min_saldo) / saldo_range)
test[saldos] = test[saldos].apply(lambda x: (x - min_saldo) / saldo_range)

minimum_saldo = min([train[saldos].replace(0,1).min().min()/2,
test[saldos].replace(0,1).min().min()/2])
train[saldos] = train[saldos].replace(0,minimum_saldo)
test[saldos] = test[saldos].replace(0,minimum_saldo)
train[saldos] = train[saldos].apply(np.log)
test[saldos] = test[saldos].apply(np.log)
```

Feature Selection

After processing the data, I have used a vanilla RandomForestClassifier to get feature importances. I have chosen only the top most important features to move on implementing and tuning other models.

```
def get_feature_importances():
    from sklearn.ensemble import RandomForestClassifier
    rf_classifier = RandomForestClassifier(random_state=1, class_weight='balanced')

    n = len(train.columns)-1

    X = train
    y = target
    rf_classifier.fit(X, y)

    i_rf = rf_classifier.feature_importances_

    importances = pd.DataFrame({'i': i_rf}, index=train.columns)
    importances.sort_values('i', ascending=False, inplace=True)

    return importances
```

```
importances = get_feature_importances()

importances = importances[importances.i>0]
most_important = importances.head(50).index.tolist()

train = train[most_important]
test = test[most_important]
```

Implementation

The implementation was simple because I'm using sklearn to import almost every model, except for xgboost, and also to cross validate them using the scoring method as 'roc_auc'.

To implement the baseline models, I have used the "stock version" of each model, changing only one parameter whenever possible: the class weight. I did this because the unbalance of our targets is one of the most relevant characteristics of the dataset.

The following code has been used to get baselines' scores:

```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from xgboost import XGBClassifier

lr_baseline = LogisticRegression('class_weight': {0:0.2, 1:0.8})
rf_baseline = RandomForestClassifier('class_weight': 'balanced')
xgb_baseline = XGBClassifier()

print(cross_val_score(lr_baseline, X, y, cv=3, scoring='roc_auc').mean())
print(cross_val_score(rf_baseline, X, y, cv=3, scoring='roc_auc').mean())
print(cross_val_score(xgb_baseline, X, y, cv=3, scoring='roc_auc').mean())
```

Refinement

The Random Forest Classifier and XGBoost model gave the best results in the initial solution, with a cross validation score of 0.8158 and 0.8383 respectively. I have used sklearn's implementation of grid search to decide the best hyper parameters tuning for these models.

The code below has been used to tune the RandomForestClassifier:

```
rf_clf = RandomForestClassifier(random_state=1, class_weight = 'balanced', n_estimators =
1000, n_jobs=-1)

rf_parameters = {'max_depth': [3, 5, 10, 20],
                 'min_samples_leaf': [1, 5, 50],
                 'min_samples_split': [2, 100, 1000, 3000],
                 'n_estimators': [10, 100, 1000, 3000]}

rf_grid_obj = GridSearchCV(rf_clf, rf_parameters, scoring='roc_auc', verbose=1, cv=3)
rf_grid_obj = rf_grid_obj.fit(train, target)
rf_opt =  rf_grid_obj.best_estimator_
print(rf_grid_obj.best_score_)
rf_opt
```

The best parameter set chosen for the Random Forest Classifier were max_depth=20, min_samples_leaf=1, min_samples_split=1000 and 1000 estimators. The cross validation score went from 0.8158 to 0.8356 after implementing grid search and optimizing the model.

To tune the XGBClassifier I wrote the following code:

```
xgb_clf = XGBClassifier(n_jobs=-1, silent=False, n_estimators=100)
xgb_parameters = {'base_score': [0.1, 0.5],
                  'max_depth': [3, 6],
                  'learning_rate': [0.01, 0.1]}

xgb_grid_obj = GridSearchCV(xgb_clf, xgb_parameters, scoring='roc_auc', verbose=1, cv=3)

xgb_grid_obj = xgb_grid_obj.fit(train, target)
xgb_opt =  xgb_grid_obj.best_estimator_
print(xgb_grid_obj.best_score_)
xgb_opt
```

The best parameters for the XGBClassifier were a base_score of 0.5, a max_depth of 3 and learning_rate equals to 0.1. The parameter tuning did not improve the model and the final XGB Classifier had a score of 0.8383.

IV. Results

Model Evaluation and Validation

The predictions submitted to Kaggle were based on datasets never seen by our model during training. These predictions scored 0.816 in the public score and 0.804 in the private score - a little lower than the ones we have acquired with training data - showing that there's some overfitting happening. It is still more than 95% of the scoring on training data, so we can assume that the predictions made upon the test set validates the model robustness.

Although a score of 0.8034 doesn't reach the top tiers in kaggle ranking for this competition, the model is fairly aligned with the solution expectation and its performance is reasonable when predicting based on unseen data, showing the capability of generalizing well.

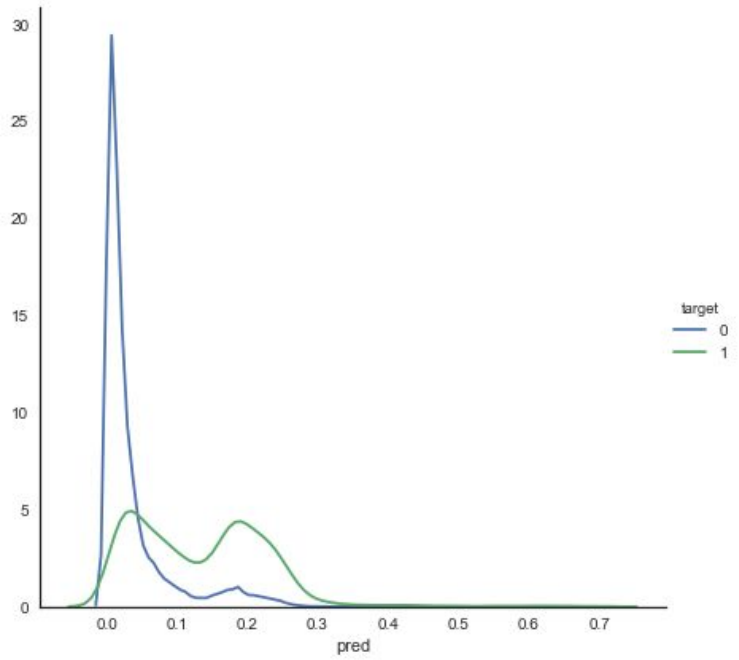
Justification

The final model had a cross validation score of 0.8034, which is way beyond the benchmark of 0.5 suggested on Kaggle. In regard to the real application of the model, I would argue that a relatively low threshold should be used, favoring recall instead of accuracy. This means that although some actions would be unnecessarily taken to already satisfied customers, the risk of losing unsatisfied ones would be drastically reduced. A threshold of 0.05 for example, would have almost 75% of unsatisfied customers flagged and actions would have been taken in order to keep this customers.

V. Conclusion

Free-Form Visualization

To check the visualize predictions being made by the model, I plotted the distribution of probabilities for training data labeled 0 and 1, separately. The visualization shows something really interesting, regarding the nature of the dataset.



As pointed out on the competition description, unhappy customers rarely voice their dissatisfaction and it might be the case that some unhappy customers are labeled 0 because they haven't responded to a given satisfaction survey, or even worse, they might have not been transparent and honest.

Our model is looking at data that corresponds to a customer behaviour and represent their relationship with the bank. If this behaviour seems to be the one of a unhappy customer, our model we'll try to point it out. When looking at the distribution of customers labeled 0, or satisfied, we see that the model is predicting a relatively high probability of unhappiness to a portion of these customers. Although it's being treated

as a wrong guess, we may wonder if the model is telling the true story, while the labels could be the ones lying to us.

Reflection

Running through this project has been extremely helpful to practice the contents of the course. Although being a binary classifier problem may sound like it was limited exercise, having hundreds of anonymized features made me think and try several machine learning techniques to preprocess the data.

In my opinion, the solution to this problem can be summarized into the following parts:

1. Data cleaning - removing useless features and replacing weird values.
2. Data exploring - By exploring the data I practiced the use of seaborn visualization library and noticed that the name of the variables had some information that I could use.
3. Feature engineering - I've added some features calculated over columns axis and created combined PCA variables to reduce data dimensionality.
4. Feature scaling - I have used MinMax scaling and log scaling to prepare data to be consumed by the algorithms.
5. Feature selection - I dropped some features that could be calculated by other features relating to the same variable and then selected the 50 most important features based on a random forest classifier feature_importance method.
6. Model selection and optimization - I have used sklearn's cross validation and grid search implementations to train and tune logistic regressions, random forest classifiers and XGBoost Classifiers.

The final model and solution fit my expectations for the problem and will be used as a start point to deal with future classification problems I need to solve.

Improvement

I came across the field of Automated Machine Learning during the project and this is something that could improve my model selection and parameter optimization.

I also wish I had tried more methods for combining different models and predictions. I read about Stacking and Voting but did not implemented it because I still need to dive a little further.

I tried to implement undersampling but it didn’t make any difference. It seemed to have improved some people’s algorithms and it’s something I will invest a little more time in another opportunity.

I believe that with more time invested and using my final solution as a new benchmark I would reach an even better solution.