Santander Customer Transaction Prediction

Submitted by - Jacob Braun, Tanay Dhaka, Akshita Srivastava, Yujun Wang, Zecong Zhao

University of Minnesota
Driven to Discover®

Table of Contents

Background and Problem Statement

Santander is a Spanish multinational corporation bank and financial-based company. They operate throughout Europe, Asia, North America, and South America. Santander seeks to continually help its customers understand their financial health and discern products and services helpful for achieving their monetary goals.

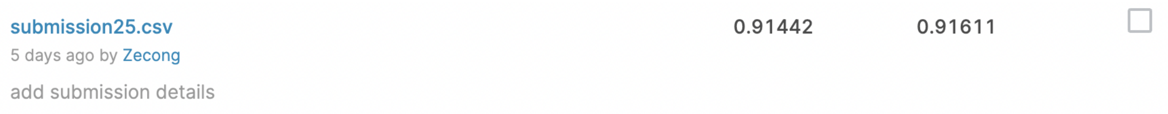
In the competition posted on Kaggle, Santander seeks help to identify which customers will make a specific transaction in the future, irrespective of the amount of money transacted. The data provided has the same structure as the real data we have available to solve this problem.

We will explore the provided anonymous data, prepare it for a model, train a model and predict the target value for the test set, then prepare a submission.

Summary/Overview

Video presentation: https://www.youtube.com/watch?v=EL_jXirPkFk

Final submission:



submission25.csv 5 days ago by Zecong add submission details	0.91442	0.91611	☐

We worked with an unbalanced dataset and implemented several alternative basic, advanced, and ensemble classification methods. Using the dataset, we discovered that selecting the right metrics is the first and most important step in determining the overall performance of the model. After trying different modeling techniques, we found out that the best results were received for ensemble 200 stacking models. In this, each base model is LightGBM with further stacking. As can be seen in the picture above, the score was 0.916.

Technical Specifications

Software and Tools

For this project, Python was used throughout for data loading, data cleaning, and model building. The following packages were utilized:

Package	Version
cv2	
Keras	2.8.0
numpy	1.21.6
os	
pandas	1.3.5
scikit-learn	1.0.2
seaborn	0.11.2
TensorFlow	2.8.0
tqdm	4.64.0
xgboost	0.9

Data Acquisition

The data was downloaded from the Kaggle competition. Below is the link:

https://www.kaggle.com/competitions/santander-customer-transaction-prediction/data

Data Description

There are two files present in the dataset:

1. train.csv - the training data
2. test.csv - the test data. There are additional rows in this file that are not included in the scoring

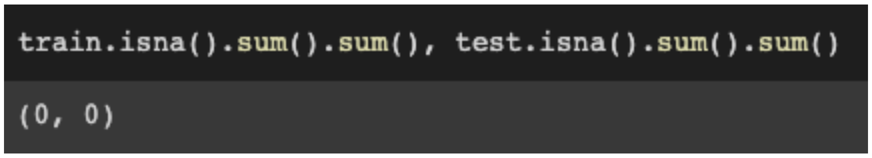
The data provided is anonymized with no identifying customer information. The dataset consists of 200,000 records, each with 200 unique features. The features are also anonymized, with arbitrary numbered names (var1, var2, ... var199). The outcome variable is binary, such that a value of 0 indicates no customer transaction and a value of 1 indicates a customer transaction. The data is unbalanced with 90% of customers not transacting. There are no missing values in the dataset and all features are numeric.

Data Exploration

Exploratory Data Analysis

Checking for missing data

After loading both the training and testing datasets, both were evaluated for missing data. There were no missing values in either dataset, so no rows were dropped or imputed.



```
train.isna().sum().sum(), test.isna().sum().sum()
```

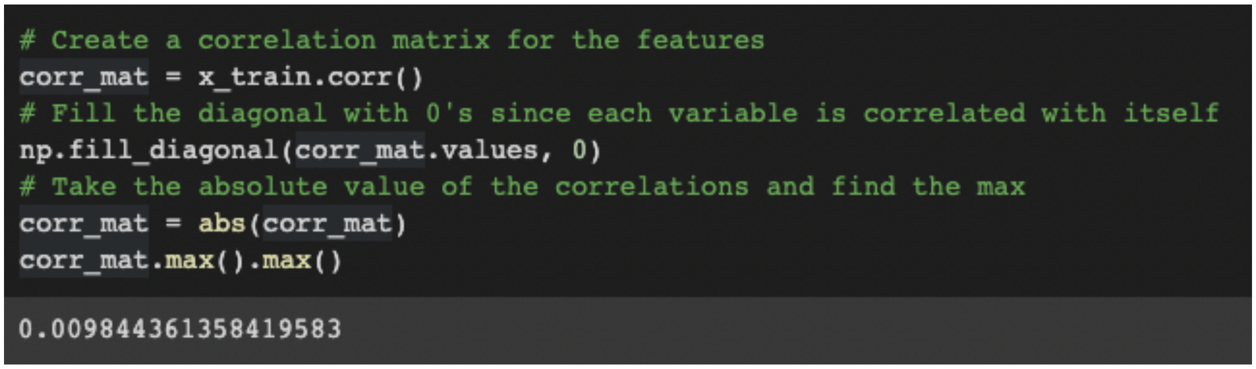
```
(0, 0)
```

Data distribution

Looking at the distribution of each variable, we found that every feature was roughly normal. It is extremely uncommon to see such clean normal distributions for all features in real-world data, so suspicions were raised about the data. The histograms are shown in Appendix Figure 1.

Correlation

Correlation matrices were generated for the features of both the training and test datasets. The values on the diagonals were then replaced with zeros since each variable is 100% correlated with itself by definition, then the maximum absolute value was found. Between both the training and testing datasets, the largest correlation between the two variables was 0.0098. This is an extremely small number and suggests no significant correlation between any features.



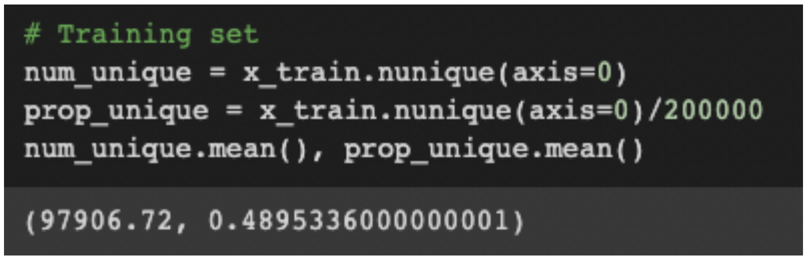
```
# Create a correlation matrix for the features
corr_mat = x_train.corr()
# Fill the diagonal with 0's since each variable is correlated with itself
np.fill_diagonal(corr_mat.values, 0)
# Take the absolute value of the correlations and find the max
corr_mat = abs(corr_mat)
corr_mat.max().max()
```

```
0.009844361358419583
```

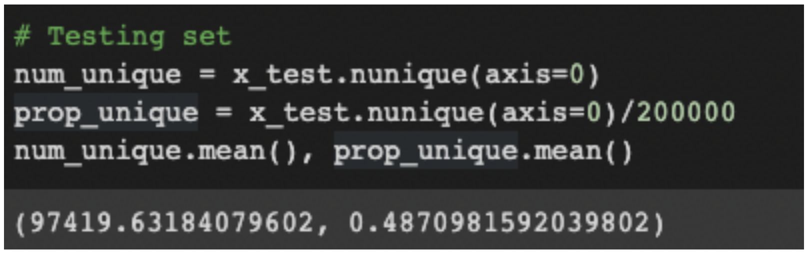
It is common for related variables relevant to a predictive problem outcome to be correlated, so the lack of correlation in this dataset raises some suspicion about the data.

Duplicate values

Having some suspicions about the data, we also checked for duplicate values within the dataset. To do so, we calculated the number and proportion of unique values per column in both datasets. The average column in the training dataset contains roughly 98,000 unique values, or 48.95% unique values. The average column in the testing dataset contains roughly 97,400 unique values or 48.71%.



```
# Training set
num_unique = x_train.nunique(axis=0)
prop_unique = x_train.nunique(axis=0)/200000
num_unique.mean(), prop_unique.mean()

(97906.72, 0.4895336000000001)
```



```
# Testing set
num_unique = x_test.nunique(axis=0)
prop_unique = x_test.nunique(axis=0)/200000
num_unique.mean(), prop_unique.mean()

(97419.63184079602, 0.4870981592039802)
```

All features in the data are continuous variables reported to 4 decimal places, so seeing less than 50% unique values across all features is extremely suspicious.

Feature Engineering

After consulting various Kaggle forums about the suspicious EDA results, we found reports that some of the data were faked by copying records and changing random values to add noise. To account for this, we tried creating new features with 2 different methods to identify this fake data.

1. Add binary features (uniqueness)

We first created new columns in the test dataset to determine if each value was unique within its respective column.

```
# Generate new features to check if each value is unique within each original
# column

col_names = [f'var_{i}' for i in range(200)]
for column in tqdm(col_names):
  c = test[column].value_counts()
  u = c.index[c == 1]
  test[column + '_u'] = test[column].isin(u)
```

Then, we compiled these generated features into a single column which represented whether or not at least one feature in each row is unique.

```
# Add a column checking if at least one feature is unique for each row
test['unique'] = test[[column + '_u' for column in col_names]].any(axis=1)
```

Based on this column, we separated the testing data into two data frames - one for 'real' data containing at least one unique value and one for 'fake' data, which consisted entirely of non-unique values. We found that the testing data was split evenly into 100,000 real records and 100,000 fake records.

```
# Separate out real test data and fake test data
test_real = test.loc[test.unique, ['ID_code'] + col_names]
test_fake = test.loc[~test.unique, ['ID_code'] + col_names]

len(test_real), len(test_fake)

(100000, 100000)
```

Next, we combined the real testing data with the training dataset to determine if there are any more duplicated records.

```
realTrTe = pd.concat([train, test_real], axis = 0)
```

We then repeated the process of generating binary features to flag if each value is unique within its respective column, and identified additional rows of fake data.

```
# Generate another set of binary features to check if each value is unique
# within each original feature column
for column in tqdm(col_names):
  c = realTrTe[column].value_counts()
  u = c.index[c == 1]
  realTrTe[column + '_unique'] = realTrTe[column].isin(u)*1
  test_fake[column + '_unique'] = 0
```

Lastly, we separated the records back into the original training and testing splits, now with 200 additional features identifying if each value is unique or not within the dataset. We believe this should allow the predictive models to learn which values are real and which values are simply generated from real records.

```
# From the combined set, isolate the real test values
test_real = realTrTe[realTrTe['ID_code'].str.contains('test')].copy()
test_real.drop(['target'], axis=1, inplace=True)

# Create a 'train' df with the new unique identifying features
# Create a 'test' df, combining the real and fake testing features
train = realTrTe[realTrTe['ID_code'].str.contains('train')].copy()
test = pd.concat([test_real, test_fake], axis=0)
```

2. Add numeric features (frequency)

Another way we did feature engineering is by counting the frequency of one value among all values of the column it is located in, instead of using a binary variable of unique or not. Here, we still combine train and real test data to calculate frequencies.

For example, for var_0, value: 8.9255 shows 8 times in the combined dataset and it will be marked as 8 in the new feature “var_0_FE”.

```
train[train['var_0']==8.9255][['ID_code','target','var_0','var_0_FE']]
```

```
             ID_code  target   var_0  var_0_FE
     0       train_0     0.0  8.9255         8
   168     train_168     0.0  8.9255         8
 74116   train_74116     0.0  8.9255         8
127906  train_127906     0.0  8.9255         8
169290  train_169290     0.0  8.9255         8
196134  train_196134     0.0  8.9255         8
```

Model Exploration

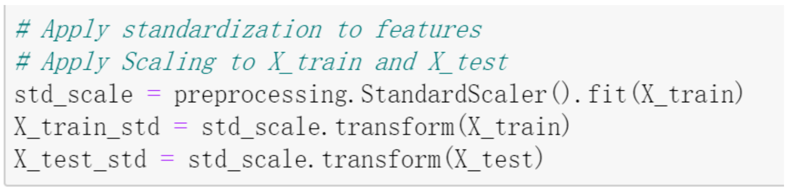
We tried to explore different modeling techniques, starting from basic models to more complex models (ensemble, stacking, etc). Below is the summary of preprocessing before model exploration:

Preprocessing	**Details**	**Results**
Discover fake data	No correlation + duplicate values	Half of the test data is suspicious
Add features	Add 200 binary features (fake or not)	Basic lgbm model score: 0.87287
Add features	Add 200 numeric features (frequency of suspicious features)	Basic lgbm model score: 0.87365
~~Normalization (fail)~~	Scale	Basic lgbm model score: 0.69496

Now we will discuss briefly different models and techniques we explored:

Normalization

We did normalization on 200 original features + 200 added frequency features, but the score dropped a lot.

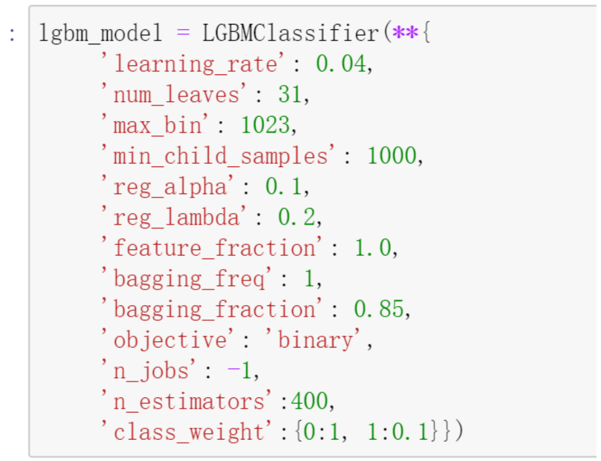


```
# Apply standardization to features
# Apply Scaling to X_train and X_test
std_scale = preprocessing.StandardScaler().fit(X_train)
X_train_std = std_scale.transform(X_train)
X_test_std = std_scale.transform(X_test)
```

Boosting

We tried different boosting models, including LightGBM, XGBoost, and CatBoost.

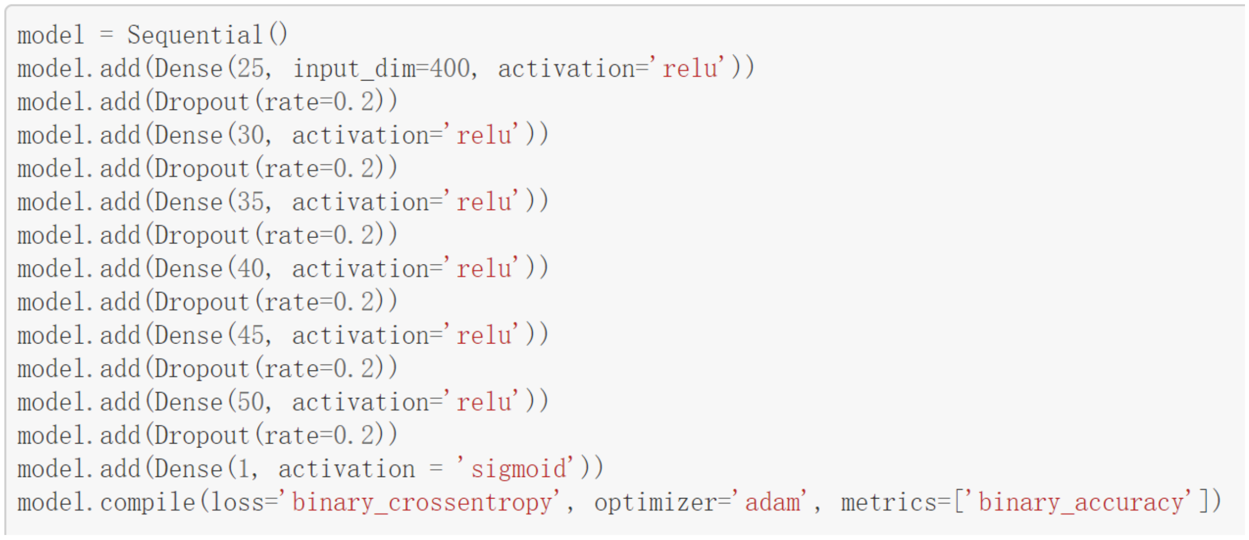
LGBM achieved 0.89484 with hyperparameters set as below.



```
: lgbm_model = LGBMClassifier(**{
        'learning_rate': 0.04,
        'num_leaves': 31,
        'max_bin': 1023,
        'min_child_samples': 1000,
        'reg_alpha': 0.1,
        'reg_lambda': 0.2,
        'feature_fraction': 1.0,
        'bagging_freq': 1,
        'bagging_fraction': 0.85,
        'objective': 'binary',
        'n_jobs': -1,
        'n_estimators':400,
        'class_weight':{0:1, 1:0.1}})
```

Neural Network

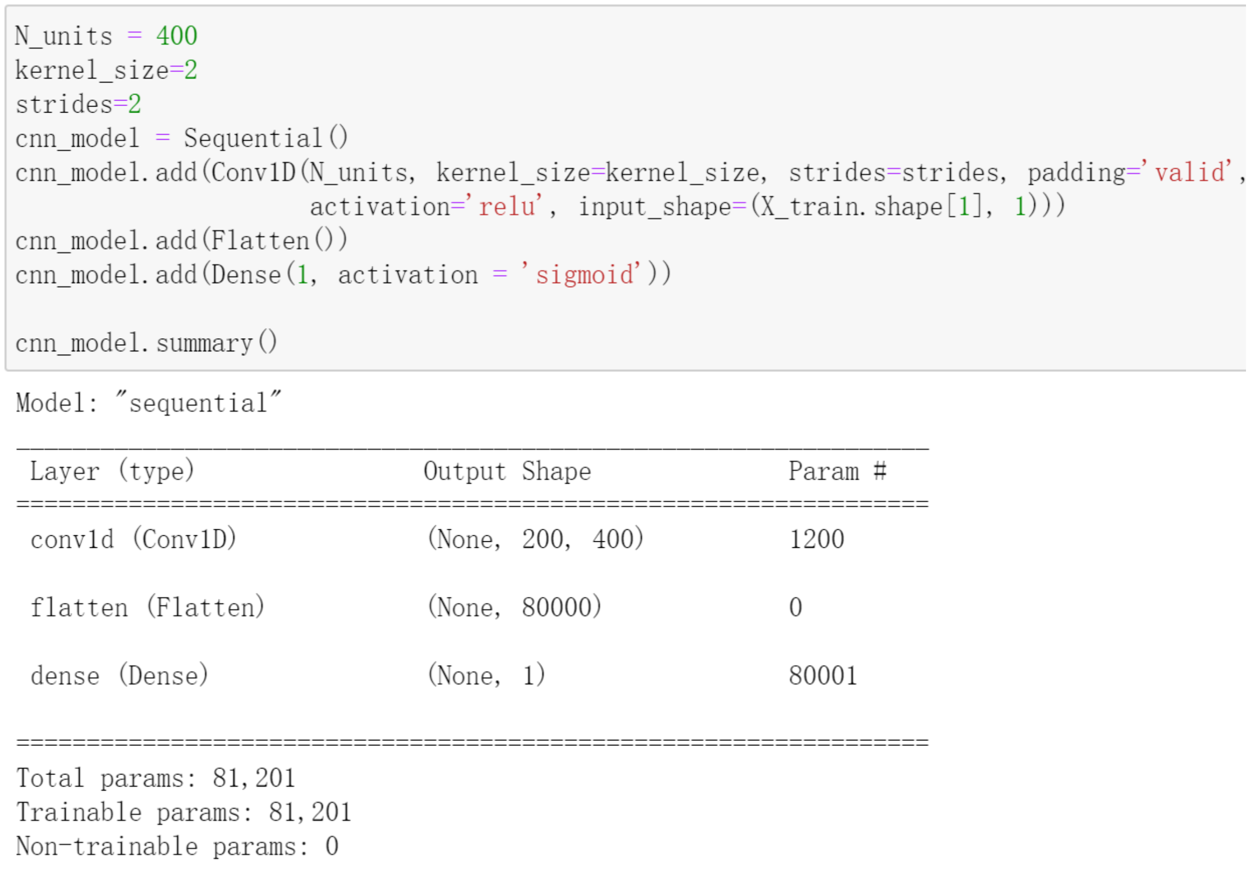
First, we tried a 13-layer neural network to train on our dataset, but the performance was not ideal, only 0.85.



```
model = Sequential()
model.add(Dense(25, input_dim=400, activation='relu'))
model.add(Dropout(rate=0.2))
model.add(Dense(30, activation='relu'))
model.add(Dropout(rate=0.2))
model.add(Dense(35, activation='relu'))
model.add(Dropout(rate=0.2))
model.add(Dense(40, activation='relu'))
model.add(Dropout(rate=0.2))
model.add(Dense(45, activation='relu'))
model.add(Dropout(rate=0.2))
model.add(Dense(50, activation='relu'))
model.add(Dropout(rate=0.2))
model.add(Dense(1, activation = 'sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['binary_accuracy'])
```

Then we introduced the idea of a Convolution Neural Network. By setting kernel size = 2 and strides = 2, we want to tell the model to find the general pattern of each pair of features(var_x & var_x_FE/var_x_unique). The performance did improve, but we just got 0.875 which is possibly due to a false learned correlation between pairs of features. Within our CNN model, we just set 3 layers and added 10 epochs, EarlyStop, and callback for a more accurate result.

The running process was very time-consuming.



```
N_units = 400
kernel_size=2
strides=2
cnn_model = Sequential()
cnn_model.add(Conv1D(N_units, kernel_size=kernel_size, strides=strides, padding='valid',
                     activation='relu', input_shape=(X_train.shape[1], 1)))
cnn_model.add(Flatten())
cnn_model.add(Dense(1, activation = 'sigmoid'))

cnn_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d (Conv1D)             (None, 200, 400)          1200

 flatten (Flatten)           (None, 80000)             0

 dense (Dense)               (None, 1)                 80001

=================================================================
Total params: 81,201
Trainable params: 81,201
Non-trainable params: 0
```

Stacking

We tried implementing several stacking models consisting of different basic models. The best result we found here featured LightGBM, XGBoost, and CatBoost models as the base with logistic regression as the meta-model. Importantly, we learned that the model performed better without normalizing the data. Additionally, the stacking model performed better with the class weights balanced in the logistic regression meta-model. This makes sense since the outcome variable is very imbalanced in the dataset. We also found that allowing passthrough in the stacking model (where the final model can see the original data as well as the base models) increased the overall performance.

Ensemble 200 models

Tree-based models and neural network models will explore some interaction relationships. For example, given var_1 < 3, the probability of target variable = 1 could be regarded as higher when var_199 > 4 than var_199 <=4. Also, the neural network can learn some entangled interaction relationships. However, the features, in our case, are non-related to each other, thus the tree-based model or NN could learn many “useless” interactions, and the training process could be very time-consuming.

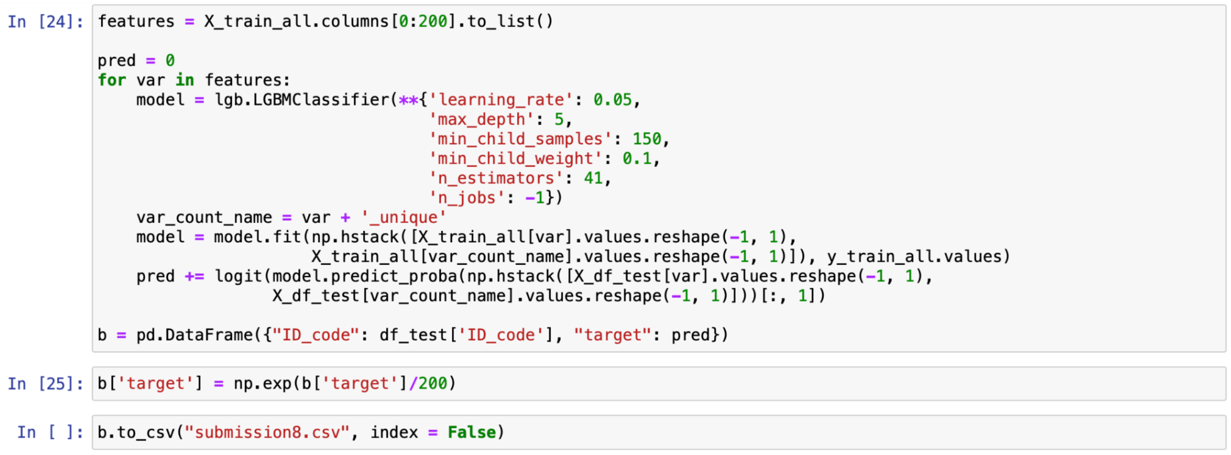
To leverage the value of original features and magic features as much as possible, we chose to use the “Ensemble 200 models” method:

(1) Each time, we build an LGBM model only with one original feature and its corresponding unique or not flag feature.

(2) Use logit to transfer the prediction of probability got from one model and sum up these results from 200 models

(3) Get mean and use exp to transfer the value back into probability

Compared with the other methods we have tried, the performance was not improved, but it provided us with a different perspective to build models which is more suitable for our case, and we started to think about combining stacking and 200 models trained on different feature pairs.

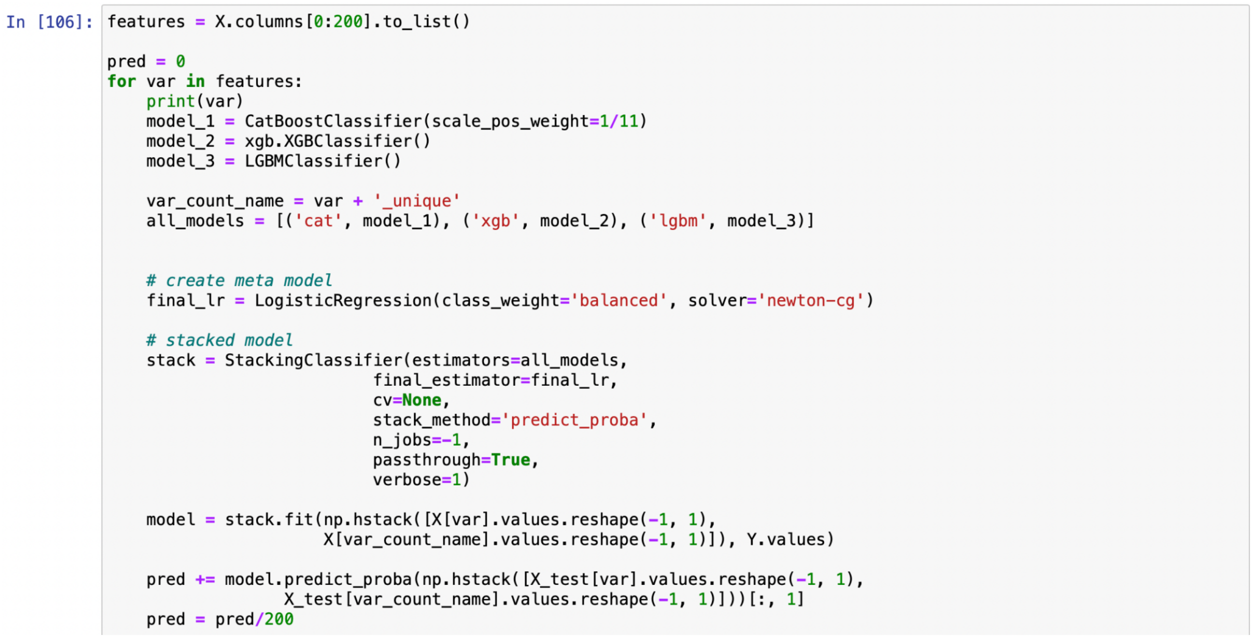


```
In [24]: features = X_train_all.columns[0:200].to_list()

         pred = 0
         for var in features:
             model = lgb.LGBMClassifier(**{'learning_rate': 0.05,
                                          'max_depth': 5,
                                          'min_child_samples': 150,
                                          'min_child_weight': 0.1,
                                          'n_estimators': 41,
                                          'n_jobs': -1})
             var_count_name = var + '_unique'
             model = model.fit(np.hstack([X_train_all[var].values.reshape(-1, 1),
                                 X_train_all[var_count_name].values.reshape(-1, 1)]), y_train_all.values)
             pred += logit(model.predict_proba(np.hstack([X_df_test[var].values.reshape(-1, 1),
                             X_df_test[var_count_name].values.reshape(-1, 1)]))[:, 1])

         b = pd.DataFrame({"ID_code": df_test['ID_code'], "target": pred})

In [25]: b['target'] = np.exp(b['target']/200)

In [ ]: b.to_csv("submission8.csv", index = False)
```

Best Model: Ensemble 200 stacking models

We still built 200 models, and for each model, we only used two features -- one original feature and its unique or not flag feature. But within each model, instead of only using one boosting model, we used stacking to ensemble different boosting models which gave us pretty good results before. After trying several combinations, we got our best model by stacking CatBoost, XGBoost, and LightGBM and the best score on Kaggle is 0.91611.

It's worth mentioning that we also tried to add corresponding hyperparameters for each boosting model -- those hyperparameters performed well at our first exploration stage, but it was not the case when we ensembled 200 stacking models together -- the score dropped and it spent a lot of time to train, so we removed these hyperparameters.



```
In [106]: features = X.columns[0:200].to_list()

          pred = 0
          for var in features:
              print(var)
              model_1 = CatBoostClassifier(scale_pos_weight=1/11)
              model_2 = xgb.XGBClassifier()
              model_3 = LGBMClassifier()

              var_count_name = var + '_unique'
              all_models = [('cat', model_1), ('xgb', model_2), ('lgbm', model_3)]


              # create meta model
              final_lr = LogisticRegression(class_weight='balanced', solver='newton-cg')

              # stacked model
              stack = StackingClassifier(estimators=all_models,
                                         final_estimator=final_lr,
                                         cv=None,
                                         stack_method='predict_proba',
                                         n_jobs=-1,
                                         passthrough=True,
                                         verbose=1)

              model = stack.fit(np.hstack([X[var].values.reshape(-1, 1),
                                 X[var_count_name].values.reshape(-1, 1)]), Y.values)

              pred += model.predict_proba(np.hstack([X_test[var].values.reshape(-1, 1),
                             X_test[var_count_name].values.reshape(-1, 1)]))[:, 1]
              pred = pred/200
```

Performance Comparison for different methods

The table below shows the summary of all models we tried with the highest scores on kaggle respectively:

Method	Model	Hyperparameters tuned	Score
Boosting	lgbm	Yes	0.89484
	xgb	Yes	0.89722
	cat	Yes	0.89870
Neural network	CNN	Yes	0.87297
	NN	Yes	0.85193
Stacking	LR + xgb + random forest, meta model = LR	No	0.89278
	lgbm + xgb , meta = LR, passthrough = true	No	0.90032
	lgbm + xgb, meta = LR, passthrough = false	No	0.87319

	lgbm + xgb + random forest, meta = LR, passthrough = true	Yes	0.89434
	lgbm + xgb + cat, meta = LR, passthrough = true	No	0.90204
Ensemble 200 models	lgbm	Yes	0.88764
Ensemble 200 stacking models	lgbm + xgb, meta = lgbm	No	0.85236
	lgbm + xgb, meta = LR	No	0.91609
	logistic + random forest + xgb + lgbm, meta = LR	No	0.91583
	cat + lgbm + xgb, meta = LR	No	0.91611
	lgbm + xgb + random forest, meta = LR	Yes	0.89134

Method	**Model**	**Hyperparameters tuned**	**AUC score**
boosting	lgbm	Yes	0.89484
neural network	CNN	Yes	0.87297
stacking	lgbm + xgb + cat, meta = LR, passthrough = true	No	0.90204
Ensemble 200 models	lgbm	Yes	0.88764
Ensemble 200 stacking models	cat + lgbm + xgb, meta = LR	No	**0.91611**

Generally, we used 3 techniques: boosting, stacking and neural network. Ensemble 200 stacking models worked best with an AUC score of 0.91611 while CNN got the worst score with a long-running time. Below is the summary of different techniques used:

1. Boosting models:

a. The performance of the 3 boosting models (LightGBM, XGBoost, CatBoost) is close, higher than 0.89 but hard to break 0.9. LightGBM and CatBoost could be faster.

b. Adding parameters improved performance for boosting models

2. Stacking models:

   a. LightGBM, XGBoost, CatBoost on logistics regression stacking model achieves 0.90204 but marginal improvement of adding more models is tiny (0.0001), hard to break 0.91

   b. Stacking 3 models on the same 400 features each model work better than stacking 200 models on a 2-feature combination each model

   c. Adding parameters will damage performances of normal stacking models but works on ensemble 200 models

3. Stacking within stacking:

   a. Combining two stacking methods helps us over 0.91

   b. Adding parameters will damage the ensemble 200 stacking models

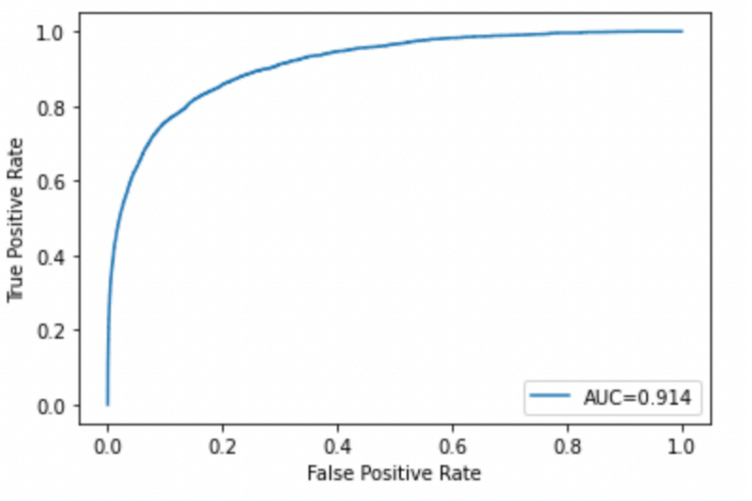
Challenges Faced

- **Realizing the existence of fake data and finding them efficiently:** By checking the distribution of each feature, correlation matrix, and calculating duplicate values within the datasets, we realized there was some noise added manually. Then after consulting the Kaggle forum, we used the criteria to detect fake data as long as there is one unique value among 200 features, then the observation will be regarded as one true data

- **Breaking 0.9 scorelines**: At the stage of model exploration, our score was always stuck at 0.89 until we used stacking, and ensemble 200 stacking models improved model performance even further
- **Normalization made the score worse:** We tried normalization on the dataset, but the score for models without normalization was way better than the normalized version
- **Neural network models cannot play a role** because of automatically learning interactions between features.
- When ensembling 200 stacking models, **adding hyperparameters didn't work** and **took a long time to train**

Business Implication

Below is the AUC curve for the best-performing model on validation data:



By implementing our model, Santander will be able to correctly rank a true purchasing customer ahead of a customer that will not actually purchase 91.4% of the time. Additionally, they will be able to identify 80% of all true purchasing customers while only mislabeling 20% of the non-purchasing customers as likely to transact. Combining these results with their own misclassification cost matrix, Santander can optimize how they target their customers.

As a next business step, Santander can devise different strategies for different types of customers using our predictive model. For customers that are predicted to make a purchase, Santander does not need to offer discounts and give away margin on a product they know will be purchased anyway. Instead, they can focus on driving additional value on the transaction through upgrades or related products. On the other hand, for customers who are predicted not to make a purchase, Santander can focus on offering discounts or other incentives to boost customer retention and increase the probability of a purchase.

For next analytical steps, we recommend Santander engage in a more robust data collection process. With the variables and customers all anonymized in this dataset, it is hard to draw any meaningful, intuitive conclusions about individual customer behavior or patterns among customer groups. By collecting data with actual feature names, Santander can apply our model to obtain more meaningful insights.

UNIVERSITY OF MINNESOTA

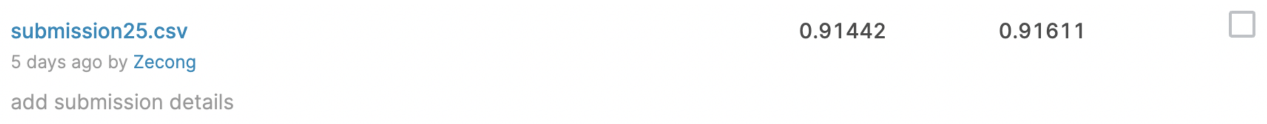
Driven to Discover®

UNIVERSITY OF MINNESOTA
Driven to Discover®

Appendix

Figure 1:

Figure 2:



submission25.csv
5 days ago by Zecong
add submission details

0.91442 0.91611

Screenshot of best score from Kaggle