Udacity Capstone Final Report

Elo Merchant Category Recommendation

Helping understand customer loyalty

I. Definition

Problem Overview

For my capstone, I will be tapping into the “Elo Merchant Category Recommendation” competition available on Kaggle to delve into a real-word problem which contains implications for a supervised modelling/regression approach to calculate customer signal.

Elo, one of the largest payment brands in Brazil, has built partnerships with merchants in order to offer promotions or discounts to cardholders. They have built machine learning models to understand the most important aspects and preferences in their customers’ lifecycle, from food to shopping. But so far none of them is specifically tailored for an individual or profile.

Provided are the available datasets for the competition:

- train.csv - the training set
- test.csv - the test set
- sample_submission.csv - a sample submission file in the correct format - contains all `card_id`s you are expected to predict for.
- historical_transactions.csv - up to 3 months' worth of historical transactions for each `card_id`
- merchants.csv - additional information about all merchants / `merchant_id`s in the dataset.
- new_merchant_transactions.csv - two months' worth of data for each `card_id` containing ALL purchases that `card_id` made at `merchant_id`s that were *not visited in the historical data*.

I will be using the train, merchant and historical transactions datasets to derive net new features before testing the outputs on the test.csv. A submission on the competition will be in the form of a card_id followed by the prediction.

Problem Statement

For my capstone, I will develop algorithms to identify and serve the most relevant opportunities to individuals, by uncovering signal in customer loyalty. This input will improve customers’ lives and help Elo reduce unwanted campaigns, to create the right experience for customers.

This process is rather complicated and a deeply explored application of machine learning for today’s companies. Through informed eda (exploratory data analysis), I will identity relationships among various features in the dataset and determine if any outliers exist. This will inform any data pre processing techniques to cleanse the data, along with encoding techniques to prepare the features for using a supervised approach.

Upon the formation of a base model, I will use additional techniques to improve this model and improve the overall score. After validating the model I will provide additional recommendations for refinement and best practices for implementing the model into production.

Metrics

Submissions are scored on the root mean squared error. RMSE(Root Mean Squared Error) is

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} (Predicted_i - Actual_i)^2}{N}}$$

defined as: where y^ is the predicted loyalty score for each card_id, and y is the actual loyalty score assigned to a card_id.

The RMSE performs well and is widely used in machine learning because the loss function in terms of RMSE is smoothly differentiable and make it easier to perform mathematical operations. Advantages of using RMSE is that it's able to penalize large errors more effectively when compared to other metrics like MAE, however from an interpretability standpoint it's much easier to use MAE.

An additional advantage is the RMSE avoids using the absolute value for the result, which can be deemed undesirable across many scenarios[1].

II. Analysis

Data Exploration & Visualization

Initial exploration will work with the training dataset to determine the distribution of the 3 features available entitled: feature_1, feature_2 and feature 3.

```
In [7]: train.head()
```

Out[7]:

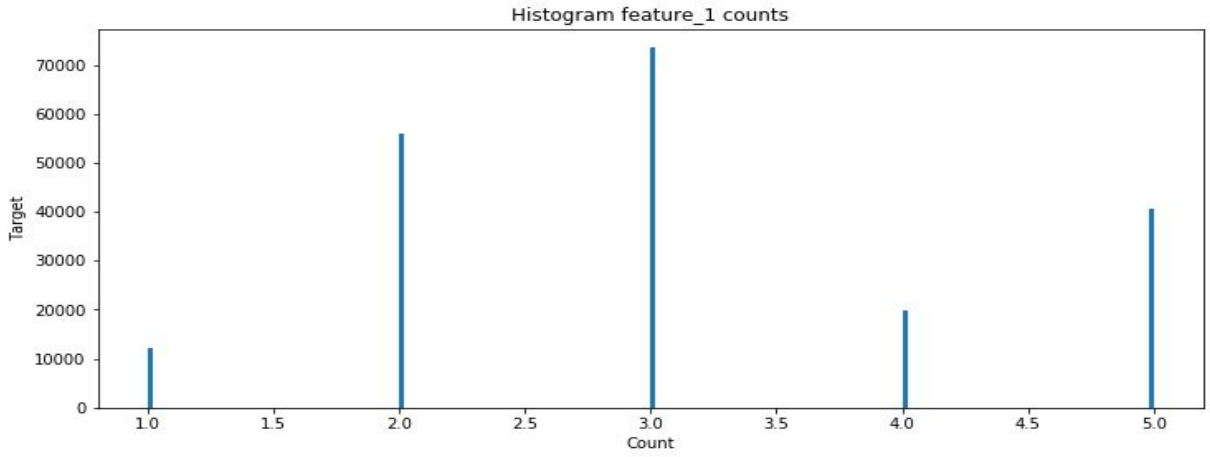
	first_active_month	card_id	feature_1	feature_2	feature_3	target
0	2017-06	C_ID_92a2005557	5	2	1	-0.820283
1	2017-01	C_ID_3d0044924f	4	1	0	0.392913
2	2016-08	C_ID_d639edf6cd	2	2	0	0.688056
3	2017-09	C_ID_186d6a6901	4	3	0	0.142495
4	2017-11	C_ID_cdbd2c0db2	1	3	0	-0.159749

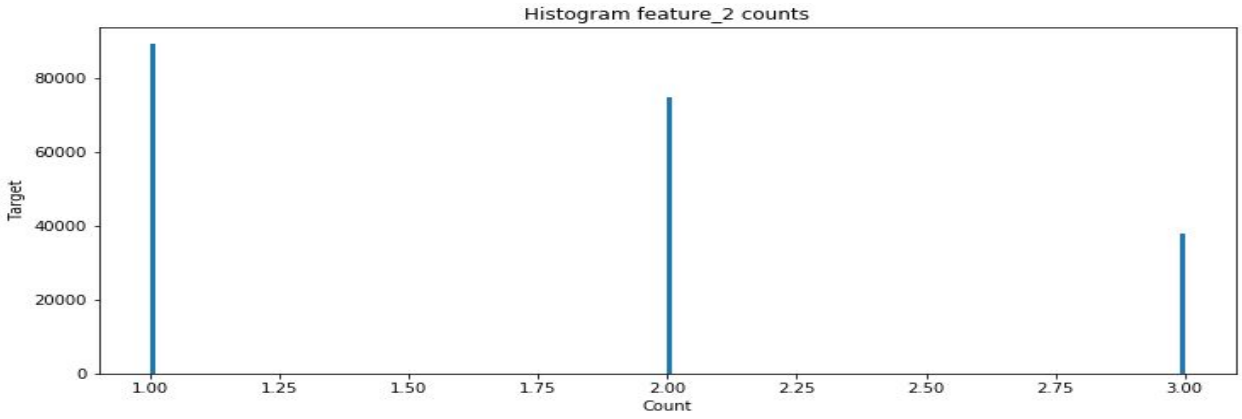
[1] MAE and RMSE — Which Metric is Better?
https://medium.com/human-in-a-machine-world/mae-and-rmse-which-metric-is-better-e60ac3bde13d

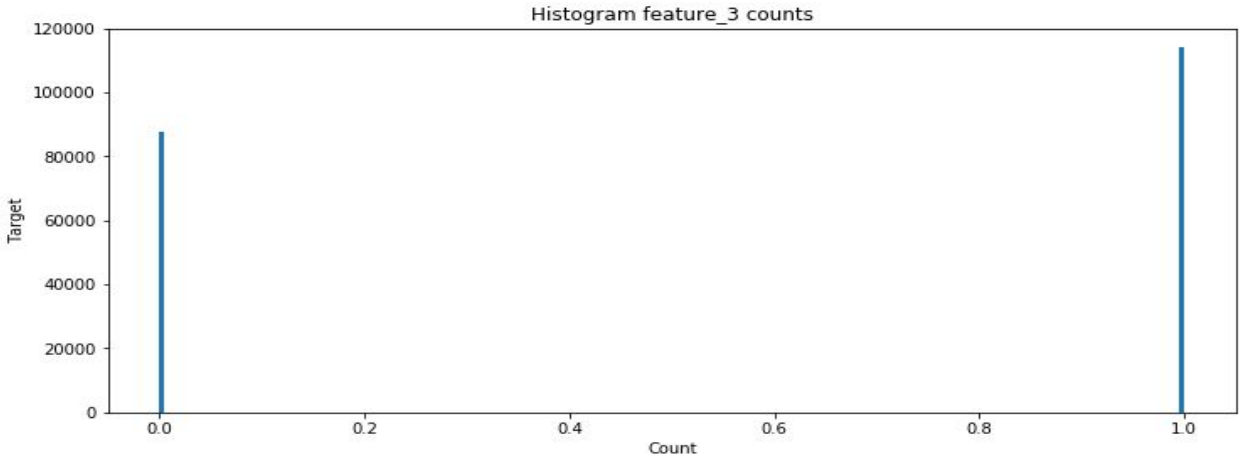
After looking at the distribution of the 3 available features, there are a few key points which stand out:

1) They are fairly evenly distributed
2) Very distinct number of values for the features
3) They are discrete

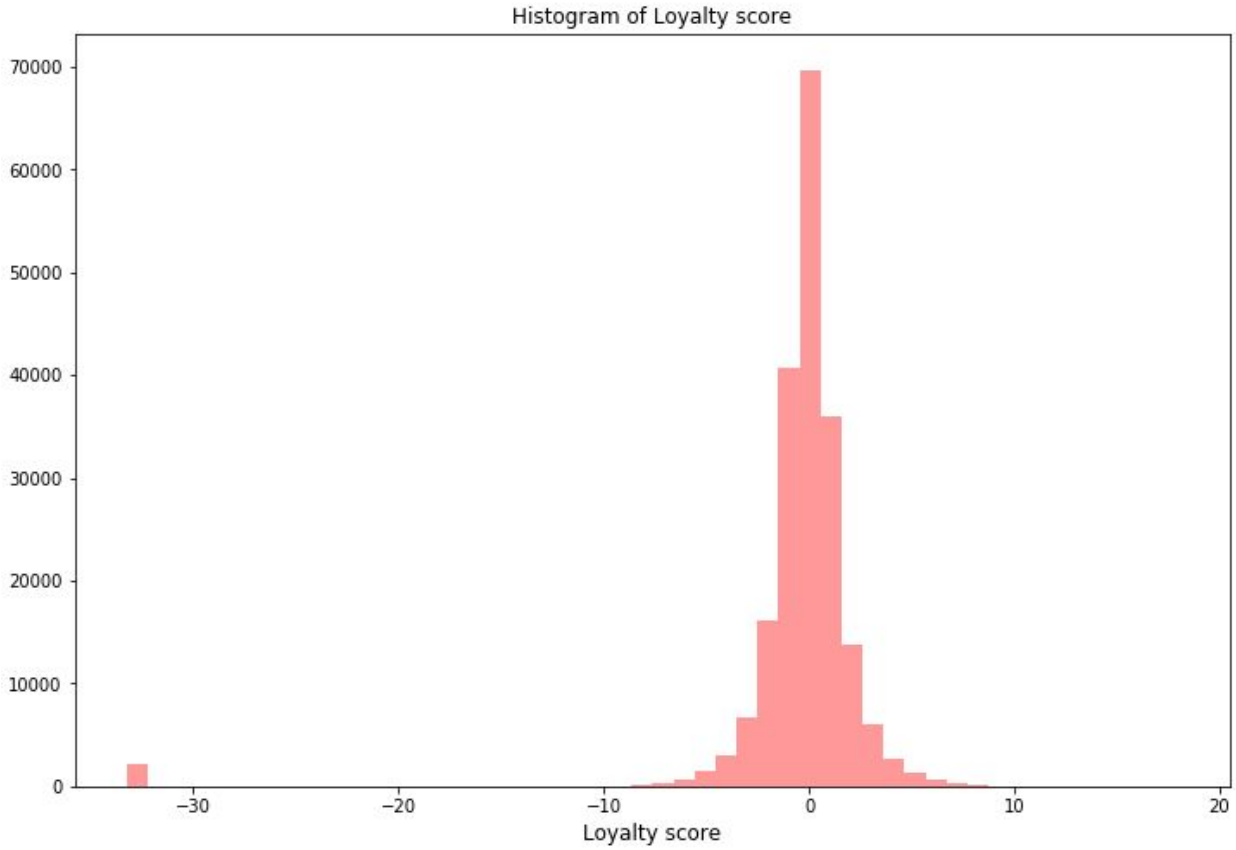
This alludes to these features being categorical and have been label-encoded.







When analyzing the distribution of the target variables, we can see that it follows a very strict normal distribution. However, we notice very few outliers exist for loyalty scores < -30. We will remove these records to ensure they do not impact the performance of the model.



The other datasets for merchants are not vital from an exploratory standpoint, but to develop net new features to be used alongside the existing features of the training dataset.

```
In [5]: train.describe(include='all')
```

Out[5]:

	first_active_month	card_id	feature_1	feature_2	feature_3	target
count	201917	201917	201917.000000	201917.000000	201917.000000	201917.000000
unique	75	201917	NaN	NaN	NaN	NaN
top	2017-09	C_ID_4135abd68d	NaN	NaN	NaN	NaN
freq	13878	1	NaN	NaN	NaN	NaN
mean	NaN	NaN	3.105311	1.745410	0.565569	-0.393636
std	NaN	NaN	1.186160	0.751362	0.495683	3.850500
min	NaN	NaN	1.000000	1.000000	0.000000	-33.219281
25%	NaN	NaN	2.000000	1.000000	0.000000	-0.883110
50%	NaN	NaN	3.000000	2.000000	1.000000	-0.023437
75%	NaN	NaN	4.000000	2.000000	1.000000	0.765453
max	NaN	NaN	5.000000	3.000000	1.000000	17.965068

Provided is the descriptive stats of the training dataset to get a further understanding of the structure of the data we're doing with before we implement our chosen baseline model.

Algorithms and Techniques

For the purposes of this project I'll be using the XGBRegressor model to predict the outcome for predicting customer loyalty defined in the target variable

The implementation of XGBoost offers several advanced features for model tuning, computing environments and algorithm enhancement. It is capable of performing the three main forms of gradient boosting (Gradient Boosting (GB), Stochastic GB and Regularized GB) and it is robust enough to support fine tuning and addition of regularization parameters.

Advantages for using XGBoost[2]:

1. **Regularization: To help reduce overfitting**
2. **Parallel Processing**
3. **High Flexibility**
4. **Handling Missing Values**
5. **Tree Pruning**
6. **Built-in Cross-Validation**
7. **Continue on Existing Model**

[2] **Complete Guide to Parameter Tuning in XGBoost (with codes in Python)**
https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/

Parameters for XGBoost

1. **General Parameters:** Guiding the overall functioning
2. **Booster Parameters**: Guide the individual booster at each step
3. **Learning Task Parameters**: Guide the optimization performed

XGBoost is an ensemble learning method which uses the aggregated predictive power of multiple learners. The result is a single model that uses the aggregated output of multiple models. The two most common form of ensemble learning fall under the realm of bagging and boosting.

Bagging: When using a decision tree for comparison, bagging will take the results of multiple decision trees to reduce the variance inherent in a single version of the model.

Boosting: For boosting, the trees are built sequentially from the previous residual results for the improvement of the model. The base learners have high bias, but gradually improve to form a final learner with lower bias and variance.

- An initial model F0 is defined to predict the target variable y. This model will be associated with a residual (y – F0)
- A new model h1 is fit to the residuals from the previous step
- Now, F0 and h1 are combined to give F1, the boosted version of F0. The mean squared error from F1 will be lower than that from F0:

$$F_1(x) <- F_0(x) + h_1(x)$$

To improve the performance of F1, we could model after the residuals of F1 and create a new model F2:

$$F_2(x) <- F_1(x) + h_2(x)$$

This can be done for *'m'* iterations, until residuals have been minimized as much as possible:

$$F_m(x) <- F_{m-1}(x) + h_m(x)$$

[3] **An End-to-End Guide to Understand the Math behind XGBoost**
https://www.analyticsvidhya.com/blog/2018/09/an-end-to-end-guide-to-understand-the-math-behind-xgboost/

For our case where the target variable is loyalty instead of salary. The tree will utilize the previous residuals to gradually reduce the error over several iterations.

During training, both the final training and validation sets after the feature engineering process are loaded into RAM with the number of threads increased to reduce overall runtime. This was an intensive model to run locally, which would warrant the approach of using a containerized Docker image to be deployed using a batch job.

Benchmark

For our baseline model we will use a simple linear regression to provide a result that we can iterate on going forward with a more sophisticated problem that best fits the problem domain.

As this problem is predicting a numerical score for customer loyalty, a regression algorithm would be a good starting point to providing a baseline measure of the root mean square error that we can iterate on.

```
: from sklearn.metrics import mean_squared_error as mse, r2_score
  # Create linear regression object
  regr = linear_model.LinearRegression()

  # Train the model using the training sets
  regr.fit(trn_x, trn_y)

  # Make predictions using the testing set
  baseline_y_pred = regr.predict(test_x)

  # The root mean squared error
  print("Root Mean squared error: %.2f"
        % np.sqrt(mse(test_y, baseline_y_pred)))

  # Explained variance score: 1 is perfect prediction
  print('Variance score: %.2f' % r2_score(test_y, baseline_y_pred))
```

```
Root Mean squared error: 3.79
Variance score: 0.03
```

As you can see the RMSE doesn't perform very well when later compared to our XG implementation discussed in the next section.

III. Methodology

Data Processing

In this step I started incorporating the additional datasets related to historical_transactions (i.e. up to 3 months worth of transactions for each card_id) along with new_merchant_transactions (i.e. two months' worth of data for each card_id containing all purchases that card_id made at merchant_ids not visited in the historical dataset).

The first step I took was to binarize any flag columns within the datasets to provide a numeric presentation of either 0 or 1 instead of 'Y' or 'N'. This was completed the binarize function.

Upon completing this, I used the get_dummies() method in pandas to convert my available categorical variables into indicator variables.

```
def binarize(df):
    for col in ['authorized_flag', 'category_1']:
        df[col] = df[col].map({'Y':1, 'N':0})
    return df

historical_transactions = binarize(historical_transactions)
new_merchant_transactions = binarize(new_merchant_transactions)

historical_transactions = pd.get_dummies(historical_transactions, columns=['category_2', 'category_3'])
new_merchant_transactions = pd.get_dummies(new_merchant_transactions, columns=['category_2', 'category_3'])

historical_transactions.head()
```

Next step we're going to compute the aggregate measures across the various input datasets (i.e. mean, median, min, max) to assess which can be included as relevant features. This is completed by using the aggregate_transactions() function.

```
def aggregate_transactions(trans, prefix):
    trans.loc[:, 'purchase_date'] = pd.DatetimeIndex(trans['purchase_date']).\
                                      astype(np.int64) * 1e-9

    agg_func = {
        'authorized_flag': ['sum', 'mean'],
        'category_1': ['mean'],
        'category_2_1.0': ['mean'],
        'category_2_2.0': ['mean'],
        'category_2_3.0': ['mean'],
        'category_2_4.0': ['mean'],
        'category_2_5.0': ['mean'],
        'category_3_A': ['mean'],
        'category_3_B': ['mean'],
        'category_3_C': ['mean'],
        'merchant_id': ['nunique'],
        'purchase_amount': ['sum', 'mean', 'max', 'min', 'std'],
        'installments': ['sum', 'mean', 'max', 'min', 'std'],
        'purchase_date': [np.ptp],
        'month_lag': ['min', 'max']
    }
    agg_trans = trans.groupby(['card_id']).agg(agg_func)
    agg_trans.columns = [prefix + '_'.join(col).strip()
                         for col in agg_trans.columns.values]
    agg_trans.reset_index(inplace=True)

    df = (trans.groupby('card_id')
          .size()
          .reset_index(name='{}transactions_count'.format(prefix)))

    agg_trans = pd.merge(df, agg_trans, on='card_id', how='left')

    return agg_trans
```

Upon completion, the result of these aggregated measures will be joined back to the training and test datasets before model training using a left join:

```
historical_transactions_agg = aggregate_transactions(historical_transactions,prefix='hist_')
new_merchant_transactions_agg = aggregate_transactions(new_merchant_transactions,prefix='new_')

train = train.merge(historical_transactions_agg,on="card_id",how="left")
test = test.merge(historical_transactions_agg,on="card_id",how="left")

train = train.merge(new_merchant_transactions_agg,on="card_id",how="left")
test = test.merge(new_merchant_transactions_agg,on="card_id",how="left")
```

To ensure a proper training process, we want to ensure that the target variable is removed from the training dataset, along with the card_id and the first_active_month.

Now our model is ready for the training process conducted by the XGBRegressor algorithm.

Implementation

For the initial approach I will be using baseline parameters for the XGBRegressor model to be further improved in the refinement section.

General Parameters:

- nthread: -1 (Number of threads for compute)
- objective: reg:linear (Linear of classifier, in our case we've chosen linear for regressor)
- learning_rate: 0.05 (Choosing a higher value between 0.05 and 3 deemed to be optimal in combination with determining the optimal number of trees)
- max_depth: 5 (Between 5-10 is optimal but we will use 5 as a starting point)
- min_child_weight: 4 (Need to typically choose a lower value for this parameter due to leaf nodes typically having smaller size groups)
- silent: 1 (To enable output for the model during run-time)
- subsample: 0.7 (Typical range is 0.5-1 as too low of a value can lead to underfitting and higher values lead to overfitting)
- Colsample_bytree: 0.7 (Denotes the fraction of columns to be randomly sampled for each tree.
- n_estimators: 500

Our implementation process is broken down into the following steps:

1) Split the training and test data using cross validation into the appropriate training, validation and test datasets
2) Instantiate the model and set the initial parameters
3) Fit the model on the existing training
4) Test the model on the test data followed by validation

Splitting the data follows simple sklearn practices of using the model_selection library of sklearn and setting the appropriate split size. For our case we're going to use a value of 0.1 and a random state of 7.

```
from sklearn.model_selection import train_test_split as tts
Trn_x,val_x,Trn_y,val_y = tts(X_train,y_train,test_size =0.1,random_state = 7)
trn_x , test_x, trn_y, test_y = tts(Trn_x , Trn_y, test_size =0.1, random_state = 7)
```

```
# converting into xgb DMatrix
Train = xgb.DMatrix(trn_x,label = trn_y)
Validation = xgb.DMatrix(val_x, label = val_y)
Test = xgb.DMatrix(test_x)
```

Setup evaluation list to calculate the RMSE for the training and validation data sets to pass to the clf parameter which instantiates the model training process.

```
history ={}  # This will record rmse score of training and test set
eval_list =[(Train,"Training"),(Validation,"Validation")]
params = xgb_grid.best_params
```

```
clf = xgb.train(params, Train, num_boost_round=119, evals=eval_list, obj=None, f
          early_stopping_rounds=40, evals_result=history)
```

After finalizing the training process, the model is then ran on the final test dataset where the RMSE is calculated and the prediction results are gathered for the submission.

```
# Checking rmse on test set (kept during data splitting)
from sklearn.metrics import mean_squared_error as mse
pred_test = clf.predict(Test)
score = mse(test_y , pred_test)
print(np.sqrt(score))
```

```
1.5589178
```

```
prediction = clf.predict(xgb.DMatrix(X_test))
df_submission=pd.DataFrame()
df_submission["card_id"] = test["card_id"].values
df_submission["target"] = np.ravel(prediction)
df_submission[["card_id","target"]].to_csv("new_submission.csv",index=False)
```

The run times for training this model were quite substantial, which was improved in the refinement process by increasing the "n_jobs" parameter for GridSearch.

The initial RMSE derived for the baseline model was 1.75, which placed within the top 75% of the Kaggle competition after the first submission.

Refinement

The refine the initial baseline model, I wanted to use an approach to optimize the hyperparameter values for the various parameters within XGBRegressor. This resulted in the use of GridSearchCV to calculate the optimal parameter values for the learning_rate and the max_depth variables for the model.

Specifying the number of jobs when initializing GridSearch also helped parallelize the workload on my local machine by spinning up 10 different applications with dedicated RAM and Cores to perform the operation. However, this bottlenecked all other processes running on my machine which would benefit from using a more robust architecture which will be addressed in the “Improvements” section.

```
xgb1 = XGBRegressor()
parameters = {'nthread':[-1], #when use hyperthread, xgboost may become slower
              'objective':['reg:linear'],
              'learning_rate': [.03, 0.05, .07], #so called `eta` value
              'max_depth': [5, 6, 7],
              'min_child_weight': [4],
              'silent': [1],
              'subsample': [0.7],
              'colsample_bytree': [0.7],
              'n_estimators': [500]}

xgb_grid = GridSearchCV(xgb1,
                        parameters,
                        cv = 3,
                        n_jobs = 10,
                        verbose=True)

xgb_grid.fit(X_train,y_train)
```

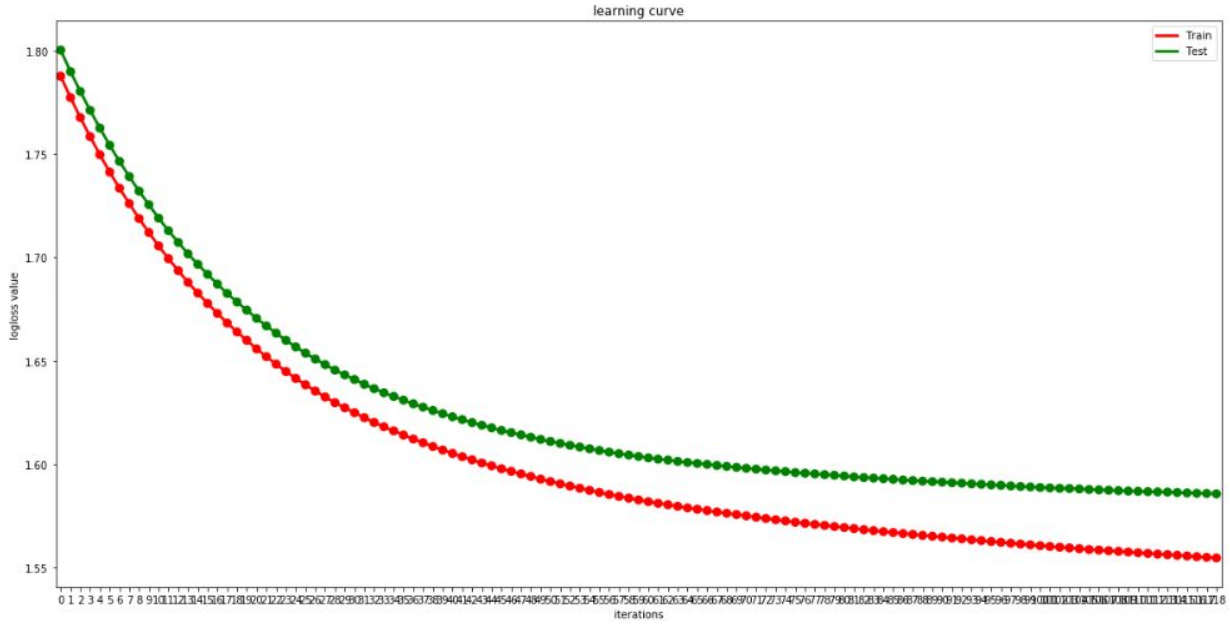
From this I was able to determine that the optimal parameter values for learning_rate and max_depth was 0.05 and 6, respectively.

Upon making these changes my training time was reduced and my RMSE improved compared to the baseline model.

Model Evaluation & Validation

My model performed better after making the necessary refinement and my RMSE was reduced overall to 1.55.

When looking at the learning rate which measures the log-loss value across multiple iterations of predictions for the test and training datasets, the model perform relatively well but towards the last few iterations you can see that the log-loss for the test dataset starts to flatten. This alludes to the model not overfitting as the test dataset curve flattens towards the final iterations. If we increase the number of iterations for this model we would see minimal benefit as the test curve is only decreasing minimally.



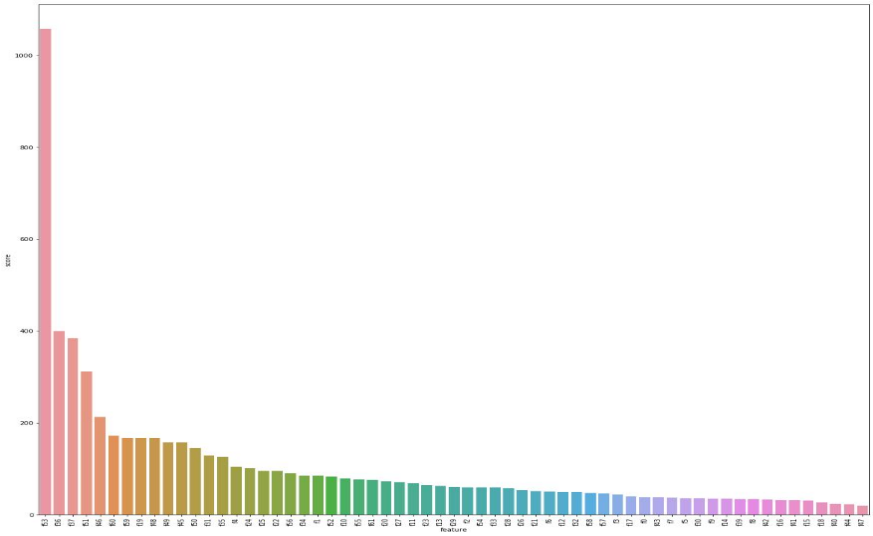
Justification

Running the model for the initial baseline of our regression tree we were able to achieve a result of 3.75 RMSE, which was improved by using GridSearch to adjust the hyperparameter values and achieve an RMSE of 1.55.

When providing the submission to Kaggle I achieved a result of the top 65%, which means a relatively strong model was developed. In summary, the application is useful but would require additional feature engineering and potentially trying out different models to achieve a better result.

Free-Form Visualization

In our case, a good way to represent the results of our implementation is to provide a feature importance plot. This can help us determine how this model can be trimmed to use only the most relevant features and drastically reduce the training time of our model.



The results show only the first 10-15 features really provide any predictive importance, whereas after that it scales down significantly and the importance values start to fall within a very small range.

Reflection

The process used for this project can be summarized using the following steps:

1. An initial problem hosted as a Kaggle Competition called Elo Merchant Category Recommendation
2. The data was downloaded and pre-processed
3. A benchmark was created for the regression use case with an XGBRegressor
4. The model was tested and results were posted to the competition

5. Improvements were made to the baseline using methods like GridSearch to optimize the hyperparameters and split up with the workload into multiple jobs

I found step 5 the most difficult as it wasn't clear what additional feature engineering could be done to improve the results of the model, in addition to the training time being significantly long.

Improvement

When looking at the result of the feature weight plot, it was evident there was only roughly 10-15 features that could be deemed to have predictive value. To reduce the training time, we could drop the features less than a specific weight or cut off that's deemed applicable.

The infrastructure hosting this workload could also be improved by dockerizing this python file and running the script with an Azure Batch instance with scaled up virtual machines. In addition, technologies such as Spark ML can be leverage to take advantage of parallelizing the dataframe operations.