Final Notebook

This notebook contain whole pipeline from data processing, featurization, model building to predicting the target values for Elo Merchant Category Recommendation.

In [1]:
```python
import pandas as pd
import numpy as np
from tqdm import tqdm
import os
import datetime
import pickle
import warnings
warnings.simplefilter("ignore")
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import Ridge
from sklearn.neighbors import KNeighborsRegressor
from joblib import dump, load
import lightgbm as lgb
import xgboost as xgb
import optuna
```

Function to reduce the memory usage by any pandas dataframe variable

In [2]:
```python
#https://www.kaggle.com/c/champs-scalar-coupling/discussion/96655
def reduce_mem_usage(df, verbose = False):

    numerics = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']
    start_mem = df.memory_usage().sum() / 1024**2
    for col in df.columns:
        col_type = df[col].dtypes
        if col_type in numerics:
            c_min = df[col].min()
            c_max = df[col].max()
            if str(col_type)[:3] == 'int':
                if c_min > np.iinfo(np.int8).min and c_max < np.iinfo(np.int8).max:
                    df[col] = df[col].astype(np.int8)
                elif c_min > np.iinfo(np.int16).min and c_max < np.iinfo(np.int16).max:
                    df[col] = df[col].astype(np.int16)
                elif c_min > np.iinfo(np.int32).min and c_max < np.iinfo(np.int32).max:
                    df[col] = df[col].astype(np.int32)
                elif c_min > np.iinfo(np.int64).min and c_max < np.iinfo(np.int64).max:
                    df[col] = df[col].astype(np.int64)
            else:
                if c_min > np.finfo(np.float16).min and c_max < np.finfo(np.float16).max:
                    df[col] = df[col].astype(np.float16)
                elif c_min > np.finfo(np.float32).min and c_max < np.finfo(np.float32).max:
                    df[col] = df[col].astype(np.float32)
                else:
                    df[col] = df[col].astype(np.float64)
    end_mem = df.memory_usage().sum() / 1024**2
    if verbose: print('Mem. usage decreased to {:5.2f} Mb ({:.1f}% reduction)'.format(end_mem, 100 *
                                                                                   (start_mem - end_mem) / s
    return df
```

Function for one hot encoding categorical columns of a dataframe.

In [3]:
```python
def onehotencoder(df, columns):
    """This function performs one hot encoding on categorical columns in a dataset and concat
    those encoded columns to the dataset and drops the original categorical columns. It takes
    dataset as dataframe object and categorical column names as list for input."""

    for col in columns:
        dummy = pd.get_dummies(df[col], prefix = col)
        df = pd.concat([df, dummy], axis = 1)
        df.drop(col, axis = 1, inplace = True)

    return df
```

Loading the Datasets

Function to load dataset

In [4]:
```python
def load_data():
    """This function loads the dataset into dataframes and returns the train, test,
    historical_transactions and new_transactions."""

    print("Loading Data.........")
    train = pd.read_csv("data/train.csv", parse_dates = ['first_active_month'])
    train = reduce_mem_usage(train)
    test = pd.read_csv("data/test.csv", parse_dates = ['first_active_month'])
    test = reduce_mem_usage(test)
    historical_transactions = pd.read_csv("data/historical_transactions.csv", parse_dates = ['purchase_date'],
                                          dtype = {"card_id" : "category"})
    historical_transactions = reduce_mem_usage(historical_transactions)
    new_transactions = pd.read_csv("data/new_merchant_transactions.csv", parse_dates = ['purchase_date'],
                                   dtype = {"card_id" : "category"})
    new_transactions = reduce_mem_usage(new_transactions)
    print("Loading of Data Completed.........")

    return train, test, historical_transactions, new_transactions
```

Processing Dataset

Function to process train and test set.

We find all the test observations that have similar feature_1, feature_2 and feature_3 values as the missing first_active_month observation and impute it with mode of the first_active_month of similar observations.

In [5]:
```python
def process_train_test(train, test):
    """This function performs dataprocessing on train and test set and returns processed train and test set.
    It takes train and test set as input."""

    print("Processing train and test set.........")
    test_null = test[test['first_active_month'].isnull()]
    test_similar = test[(test.feature_1 == test_null.feature_1.values[0]) & (test.feature_2 == test_null.featur
                   & (test.feature_3 == test_null.feature_3.values[0])]
    test.first_active_month[test['first_active_month'].isnull()] = test_similar['first_active_month'].mode()[0]
    del test_null
    del test_similar
    train.to_csv("data/train_processed.csv", index = False)
    test.to_csv("data/test_processed.csv", index = False)
    print("Processing of train and test set Completed.........")

    return train, test
```

Function to process historical and new transactions.

In [6]:
```python
def impute_merchant_id(df):
    """This function imputes the null merchant ids in historical and new transactions. It takes takes
    transaction dataset as dataframe and returns the transaction dataframe after imputing."""

    Merchants_Categorical_Columns = ["merchant_category_id", "subsector_id", "city_id", "state_id"]
    Merchants_Categorical_Dtypes = {col: "category" for col in Merchants_Categorical_Columns}
    merchants = pd.read_csv("data/merchants.csv", dtype = Merchants_Categorical_Dtypes)

    df_null = df[df['merchant_id'].isnull()]
    df_null_index = df_null.index
    for idx in tqdm(df_null_index):
        df_similar = merchants[(merchants.merchant_category_id == df_null.merchant_category_id.loc[idx]) &
                               (merchants.subsector_id == df_null.subsector_id.loc[idx]) &
                               (merchants.city_id == df_null.city_id.loc[idx])]
        if df_similar.shape[0] != 0:
            df.merchant_id.loc[idx] = df_similar['merchant_id'].mode()[0]
        del df_similar
    del df_null
    del merchants
    df['merchant_id'].fillna('NAN', inplace = True)
    df['merchant_id'] = df['merchant_id'].astype('category')

    return df
```

In [7]:
```python
def impute_category(df, null_columns, train_columns, model_prefix):
    """This function imputes the null category columns of historical and new transactions
    by training classifier model from non null columns. It takes transaction as dataframe,
    categorical columns with null values as list, non null columns as list and prefix for
    the saved model as string for input."""

    for col in null_columns:
        test_df = df.loc[df[col].isna()][train_columns]
        train_df = df.loc[df[col].notna()][train_columns]
        train_y = df.loc[df[col].notna()][col]

        path = 'data/' + model_prefix + '_' + str(col) + '_model'
        if os.path.exists(path):
            clf = pickle.load(open(path, 'rb'))
        else:
            print("Training model to impute", col)
            clf = LogisticRegression()
            clf.fit(train_df, train_y)
            pickle.dump(clf, open(path, 'wb'))
        print("Imputing predicted category from model to null values in", col)
        df.loc[df[col].isna(), col] = clf.predict(test_df)
        df[col] = df[col].astype(np.int8)
        del train_df
        del test_df
        del train_y

    return df
```

Firstly, the authorized_flag, category_1 and category_3 columns of historical transactions and new transactions are Label Encoded.

We are finding merchant_ids in merchants having similar merchant_category_id, subsector_id and city_id as null observations and impute null observations with the mode of similar merchant_ids. We are not considering state_id as any two similar city_ids will have same state_id. For remaining merchant_id null values we are simply imputing NAN.

We are imputing null values in category_2 and category_3 columns of historical and new transactions by training classifier models from non null columns of these transactions. We will use these classifier models to predict the null values in category_2 and category_3 of historical and new transactions.

It should be natural to expect the values of purchase amount to be positive which is obviously not the case here. We are using insights provided by Raddar in his notebook in Kaggle for de-anonymizing the purchase amount and transforming the purchase amount into it's observed value.

Finally, we are one hot encoding the categorical columns in historical transactions and new transactions.

In [8]:
```python
def process_transactions(historical_transactions, new_transactions):
    """This function performs dataprocessing on historical and new transactions and returns a
    combined processed transactions dataframe. It takes historical transactions and new transactions
    as input."""

    print("Processing historical and new transactions.........")
    historical_transactions['authorized_flag'] = historical_transactions['authorized_flag'].map({'Y':1,
                                                                                                  'N':0}).astype
    historical_transactions['category_1'] = historical_transactions['category_1'].map({'Y':1, 'N':0}).astype(np
    historical_transactions['category_3'] = historical_transactions['category_3'].map({'A':0, 'B':1, 'C':2})

    new_transactions['authorized_flag'] = new_transactions['authorized_flag'].map({'Y':1, 'N':0}).astype(np.int
    new_transactions['category_1'] = new_transactions['category_1'].map({'Y':1, 'N':0}).astype(np.int8)
    new_transactions['category_3'] = new_transactions['category_3'].map({'A':0, 'B':1, 'C':2})

    historical_transactions = impute_merchant_id(historical_transactions)
    new_transactions = impute_merchant_id(new_transactions)

    null_columns = ['category_2', 'category_3']
    train_columns = ['authorized_flag', 'category_1', 'installments', 'month_lag', 'purchase_amount',
                     'merchant_category_id', 'subsector_id', 'city_id', 'state_id']
    historical_transactions = impute_category(historical_transactions, null_columns, train_columns, 'historical
    new_transactions = impute_category(new_transactions, null_columns, train_columns, 'new')

    #https://www.kaggle.com/code/raddar/towards-de-anonymizing-the-data-some-insights/notebook
    historical_transactions['purchase_amount'] = ((historical_transactions['purchase_amount'].astype(np.float64
                                                   0.00150265118) + 497.06)
    new_transactions['purchase_amount'] = ((new_transactions['purchase_amount'].astype(np.float64) / 0.00150265

    categorical_columns = ['category_1', 'category_2', 'category_3']
    historical_transactions = onehotencoder(historical_transactions, categorical_columns)
    new_transactions = onehotencoder(new_transactions, categorical_columns)

    historical_transactions.to_csv("data/historical_transactions_processed.csv", index = False)
    new_transactions.to_csv("data/new_transactions_processed.csv", index = False)
    print("Processing of historical and new transactions Completed.........")

    return historical_transactions, new_transactions
```

Feature Engineering

Function to perform featurization on train and test.

We are using the reference date of 1/2/2018 to calculate the elapsed time for each card id. The elapsed time feature will indicate the number of days, the cardholder has been using the card. We will also divide first active month column into first active year and first active month categorical columns. First active year will denote the year and first active month will denote the month, the cardholder started using the card.

We will be adding outlier identification column to train set. The outlier columns will be 1 for card_ids having outlier value target and 0 for remaining card_ids. This outlier column will be used for stratified splitting of train set during model training.

In [9]:
```python
def feature_train_test(train, test):
    """This function performs featurization on train and test set. It takes train and test set as input
    and returns featurized train and test set."""

    print("Performing featurization on train and test set.........")
    train['elapsed_time'] = (datetime.date(2018, 2, 1) - train['first_active_month'].dt.date).dt.days
    train['first_active_year'] = train['first_active_month'].dt.year
    train['first_active_month'] = train['first_active_month'].dt.month

    test['elapsed_time'] = (datetime.date(2018, 2, 1) - test['first_active_month'].dt.date).dt.days
    test['first_active_year'] = test['first_active_month'].dt.year
    test['first_active_month'] = test['first_active_month'].dt.month

    train['outlier'] = 0
    train['outlier'][train['target'] > 30] = 1

    train.to_csv("data/train_featurized.csv", index = False)
    test.to_csv("data/test_featurized.csv", index = False)
    print("Featurization of train and test set Completed.........")

    return train, test
```

Function to perform featurization on transactions.

In [10]:
```python
def date_featurization(df, column):
    """This function featurize the date column of a dataframe by engineering
    new features such as year, month, day, hour etc. It takes the dataset as
    dataframe and date column as string for input and returns the dataframe
    with added features."""

    df['year'] = df[column].dt.year
    df['month'] = df[column].dt.month
    df['dayofweek'] = df[column].dt.dayofweek
    df['date'] = df[column].dt.day
    df['hour'] = df[column].dt.hour
    df['weekend'] = 0
    df['weekend'][df['dayofweek'] >= 5] = 1

    return df
```

In [11]:
```python
def agg_featurization(df, groupby, agg_dict, prefix = ""):
    """This function performs aggregation on a dataframe and returns the aggregate features
    dataframe. It takes dataset as dataframe, groupby columns on which aggregate has to be performed
    as list, aggregate functions to be performed on columns as dictionary and prefix to be added to
    aggregated feature column name."""

    agg_df = df.groupby(groupby).agg(agg_dict)
    if prefix != "":
        agg_df.columns = [prefix + '_' + '_'.join(col) for col in agg_df.columns.values]
    else:
        agg_df.columns = ['_'.join(col) for col in agg_df.columns.values]
    agg_df.reset_index(inplace = True)

    return agg_df
```

In [12]:
```python
def category_aggregate_featurization(df, columns, groupby, agg_dict, prefix = ""):
    """This function performs aggregation on a dataframe based on groupby and each categorical columns
    and returns the aggregate features dataframe. It takes dataset as dataframe, groupby columns on
    which aggregate has to be performed as list, categorical columns which have to be aggregated with
    groupby columns as list and aggregate functions to be performed on columns as dictionary."""

    df_features = pd.DataFrame(df['card_id'].unique(), columns = ['card_id'])
    for col in columns:
        agg_df = agg_featurization(df[df[col] == 1], groupby, agg_dict, prefix = prefix + "_" + col)
        df_features = pd.merge(df_features, agg_df, on = 'card_id', how = 'left')
        del agg_df

    return df_features
```

In [13]:
```python
def month_lag_aggregate_featurization(df, groupby, agg_dict, prefix = ""):
    """This function performs aggregation on a dataframe based on groupby and each value of month lag
    columns and returns the aggregate features dataframe. It takes dataset as dataframe, groupby columns on
    which aggregate has to be performed as list and aggregate functions to be performed on columns as
    dictionary."""

    df_features = pd.DataFrame(df['card_id'].unique(), columns = ['card_id'])
    for value in df['month_lag'].unique():
        agg_df = agg_featurization(df[df['month_lag'] == value], groupby, agg_dict,
                                   prefix = prefix + '_month_lag_' + str(value))
        df_features = pd.merge(df_features, agg_df, on = 'card_id', how = 'left')
        del agg_df

    return df_features
```

In [14]:
```python
def successive_agg_featurization(df, groupby1, groupby2, columns, agg_dict, prefix = ""):
    """This function performs successive aggregation on a dataframe and returns the successive aggregate
    features dataframe. It takes dataset as dataframe, groupby1 and groupby2 on which aggregate
    has to be performed as strings, columns on which the aggregate function is to be performed and
    aggregate functions to be performed on columns as dictionary."""

    intermediate_agg_df = df.groupby([groupby1, groupby2])[columns].mean()
    successive_agg_df = agg_featurization(intermediate_agg_df, groupby1, agg_dict, prefix = prefix + "_" + grou

    return successive_agg_df
```

In [15]:
```python
def RFM_Score(x, col, rfm_quantiles):
    """Function to calculate Recency, Frequency and Monetary value score based on quantiles.
    It takes respective value, column name and quantiles dataframe as input."""

    score_1 = 1
    score_2 = rfm_quantiles.shape[0]
    for i in range(rfm_quantiles.shape[0]):
        if x <= rfm_quantiles[col].values[i]:
            return score_2 if col is 'recency' else score_1
        score_1 += 1
        score_2 -= 1
```

In [16]:
```python
#https://www.kaggle.com/code/rajeshcv/customer-loyalty-based-on-rfm-analysis/notebook
def rfm_feature(df, quantiles):
    """This function performs the RFM featurization on dataset by generating the RFM score
    and RFM index. It takes dataset as dataframe, and quantile values for scoring as list and
    returns the RFM features as dataframe."""

    agg_dict = {
        'card_id'         : ['count'],
        'purchase_date'   : ['max'],
        'purchase_amount' : ['sum']
    }
    rfm_feature = agg_featurization(historical_transactions, groupby, agg_dict)
    rfm_feature['recency'] = (datetime.date(2018, 3, 1) - rfm_feature['purchase_date_max'].dt.date).dt.days
    rfm_feature.rename(columns = {'card_id_count' : 'frequency', 'purchase_amount_sum' : 'monetary_value'}, inp
    rfm_feature = rfm_feature.drop(columns = ['purchase_date_max'])

    rfm_quantiles = rfm_feature.quantile(q = quantiles)
    rfm_feature['R_score'] = rfm_feature['recency'].apply(RFM_Score, args = ('recency', rfm_quantiles))
    rfm_feature['F_score'] = rfm_feature['frequency'].apply(RFM_Score, args = ('frequency', rfm_quantiles))
    rfm_feature['M_score'] = rfm_feature['monetary_value'].apply(RFM_Score, args = ('monetary_value', rfm_quant
    rfm_feature['RFM_Score'] = rfm_feature['R_score'] + rfm_feature['F_score'] + rfm_feature['M_score']
    rfm_feature['RFM_index'] = rfm_feature['R_score'].map(str) + rfm_feature['F_score'].map(str) + rfm_feature[
    rfm_feature['RFM_index'] = rfm_feature['RFM_index'].astype(int)
    rfm_feature = rfm_feature.drop(columns = ['recency', 'frequency', 'monetary_value'])

    return rfm_feature
```

We are engineering new columns from purchase date such as purchase year, month, weekday, date etc. These columns will be used to generate time related features.

Then we are engineering count of historical and new transactions features. These features will indicate the number of historical and new transactions done by each card id.

We are performing aggregation of card_id's to find different features for all historical and new transactions columns.

1. authorized_flag features will indicate total authorized transactions and percentage of authorized transactions done by each card_id.
2. category_1, category_2 and category_3 features will indicate total and percentage of transactions for that particular category value done by each card_id.
3. merchant_id, merchant_category_id, subsector_id, city_id and state_id features will indicate number of unique merchants, merchant categories, subsectors, cities and states, each card_id did transaction at.
4. month_lag features will indicate minimum and maximum transaction lag from reference date and and recency of transaction for each card_id.
5. purchase date features will indicate the oldest and the newest date of transactions done by each card_id.
6. year, month, dayofweek, date, hour features will indicate number of unique, mean, maximum and minimum of years, months, day of weeks, dates and hours, during which each card_id did transactions.
7. weekend feature will indicate total and percentage of transactions done by each card_id on weekend.
8. purchase_amount features will indicate total, average, maximum, minimum and variance of the amount spent by each card_id.
9. installments features will indicate number of unique, total, average, maximum and minimum number of installments for transactions done by each card_id.

We are also engineering some additional features from generated aggregate features.

1. The difference in historical transaction count and historical authorized flag sum date will indicate the number of declined historical transactions for by cardholders.
2. The difference in maximum purchase date and minimum purchase date will indicate the duration in days when the transactions were done by each cardholders.
3. The ratio of total purchase amount and duration of transactions will indicate the purchase amount spent per day by each cardholders.
4. The difference in maximum purchase amount and minimum purchase amount will indicate the range of amount spent by each cardholders.
5. The ratio of transaction count and duration of transactions will indicate the transactions done per day by each cardholders.
6. The ratio of transaction count and unique number of merchant ids will indicate the transactions done per merchant by each cardholders.
7. The ratio of transaction count and unique number of city ids will indicate the transactions done per city by each cardholders.
8. The ratio of transaction count and unique number of state ids will indicate the transactions done per state by each cardholders.
9. The ratio of transaction count and unique merchant category ids will indicate the transactions done per merchant category by each cardholders.

During EDA, we found that purchase amount had different distributions for different values of category 1, category 2 and category 3. We are performing aggregation of card id with different values of category 1, category 2 and category 3 columns and find features for purchase amount.

We are also performing aggregation of card id for each month lag column values to find features for purchase amount. These features will indicate month wise features of purchase amount.

We are performing aggregation of card id and installments columns to find features for authorized flag and purchase amount. These features will indicate installment wise features of authorized flag and purchase amount.

RFM is a market research tool for customer segmentation based on customer value to the firm. R stands for Recency, F for Frequency and M for Monetary value. Recency is the number of days since last purchase, Frequency is the total number of purchases and Monetary Value is the total money, the customer spent. An RFM analysis evaluates customers by scoring them in three categories: how recently they've made a purchase, how often they buy, and the size of their purchases. Based on target values, we will find quantiles which will be used to calculate scores for each card id. The card ids will be scored based on which quantile their recency, frequency and monetary values fall into. Also, recency will be scored opposite of frequency and monetary value i.e., smaller the recency value higher the score whereas larger the frequency and monetary values higher the score. The RFM score is the sum of the recency score, frequency score and monetary score while the RFM index is obtained by combining the recency score, frequency score and monetary score.

In [17]:
```python
def feature_transactions(historical_transactions, new_transactions):
    """This function performs featurization on transactions dataset. It takes
    transactions dataset as input and returns engineered features for each
    card id as dataframe."""

    print("Performing featurization on historical and new transactions.........")
    historical_transactions = date_featurization(historical_transactions, 'purchase_date')
    new_transactions = date_featurization(new_transactions, 'purchase_date')

    hist_transactions_features = historical_transactions.groupby(['card_id']).size().reset_index()
    hist_transactions_features.columns = ['card_id', 'hist_transc_count']

    new_transactions_features = new_transactions.groupby(['card_id']).size().reset_index()
    new_transactions_features.columns = ['card_id', 'new_transc_count']

    groupby = ['card_id']
    agg_dict = {
    'authorized_flag' : ['sum', 'mean'],
    'category_1_0'    : ['sum', 'mean'],
    'category_1_1'    : ['sum', 'mean'],
    'category_2_1'    : ['sum', 'mean'],
    'category_2_2'    : ['sum', 'mean'],
    'category_2_3'    : ['sum', 'mean'],
    'category_2_4'    : ['sum', 'mean'],
    'category_2_5'    : ['sum', 'mean'],
    'category_3_0'    : ['sum', 'mean'],
    'category_3_1'    : ['sum', 'mean'],
    'category_3_2'    : ['sum', 'mean'],

    'merchant_id'          : ['nunique'],
    'merchant_category_id': ['nunique'],
    'subsector_id'         : ['nunique'],
    'city_id'              : ['nunique'],
    'state_id'             : ['nunique'],

    'month_lag'    : ['min', 'max', 'mean'],
    'purchase_date': ['min', 'max'],
    'year'         : ['nunique', 'mean', 'min', 'max'],
    'month'        : ['nunique', 'mean', 'min', 'max'],
    'dayofweek'    : ['nunique', 'mean', 'min', 'max'],
    'date'         : ['nunique', 'mean', 'min', 'max'],
    'hour'         : ['nunique', 'mean', 'min', 'max'],
    'weekend'      : ['sum', 'mean'],

    'purchase_amount': ['sum', 'mean', 'max', 'min', 'std'],
    'installments'   : ['nunique', 'sum', 'mean', 'max', 'min']
    }

    hist_transactions_features = pd.merge(hist_transactions_features,
                                          agg_featurization(historical_transactions, groupby, agg_dict, prefix = 'h
                                          on = 'card_id', how = 'left')
    del agg_dict['authorized_flag']
    new_transactions_features = pd.merge(new_transactions_features,
                                         agg_featurization(new_transactions, groupby, agg_dict, prefix = 'new'),
                                         on = 'card_id', how = 'left')

    hist_transactions_features['hist_denied_count'] = (hist_transactions_features['hist_transc_count'] -
                                                       hist_transactions_features['hist_authorized_flag_sum'])
    hist_transactions_features['hist_transaction_days'] = (hist_transactions_features['hist_purchase_date_max']
                                                           hist_transactions_features['hist_purchase_date_min']).dt
    hist_transactions_features['hist_purchase_amount_per_day'] = (hist_transactions_features['hist_purchase_amo
                                                                  (1 + hist_transactions_features['hist_transaction
    hist_transactions_features['hist_purchase_amount_diff'] = (hist_transactions_features['hist_purchase_amount
                                                               hist_transactions_features['hist_purchase_amount_min
    hist_transactions_features['hist_transactions_per_day'] = (hist_transactions_features['hist_transc_count']
                                                               (1 + hist_transactions_features['hist_transaction_da
    hist_transactions_features['hist_transactions_per_merchant_id'] = (hist_transactions_features['hist_transc_
                                                                       (1 + hist_transactions_features['hist_merchant_
    hist_transactions_features['hist_transactions_per_city_id'] = (hist_transactions_features['hist_transc_coun
                                                                   (1 + hist_transactions_features['hist_city_id_nu
    hist_transactions_features['hist_transactions_per_state_id'] = (hist_transactions_features['hist_transc_cou
                                                                    (1 + hist_transactions_features['hist_state_id_
    hist_transactions_features['hist_transactions_per_merchant_category_id'] =\
    (hist_transactions_features['hist_transc_count'] / (1 + hist_transactions_features['hist_merchant_category_
    hist_transactions_features = hist_transactions_features.drop(columns = ['hist_purchase_date_max',
                                                                            'hist_purchase_date_min'])
    hist_transactions_features = reduce_mem_usage(hist_transactions_features)

    new_transactions_features['new_transaction_days'] = (new_transactions_features['new_purchase_date_max'] -
                                                         new_transactions_features['new_purchase_date_min']).dt.day
    new_transactions_features['new_purchase_amount_per_day'] = (new_transactions_features['new_purchase_amount_
                                                                (1 + new_transactions_features['new_transaction_day
    new_transactions_features['new_purchase_amount_diff'] = (new_transactions_features['new_purchase_amount_max
                                                             new_transactions_features['new_purchase_amount_min'])
    new_transactions_features['new_transaction_per_day'] = (new_transactions_features['new_transc_count'] /
                                                            (1 + new_transactions_features['new_transaction_days'])
    new_transactions_features['new_transactions_per_merchant_id'] = (new_transactions_features['new_transc_coun
                                                                     (1 + new_transactions_features['new_merchant_i
    new_transactions_features['new_transactions_per_city_id'] = (new_transactions_features['new_transc_count']
                                                                 (1 + new_transactions_features['new_city_id_nuniqu
    new_transactions_features['new_transactions_per_state_id'] = (new_transactions_features['new_transc_count']
                                                                  (1 + new_transactions_features['new_state_id_nuni
    new_transactions_features['new_transactions_per_merchant_category_id'] = (new_transactions_features['new_tr
                                                                             (1 + new_transactions_features['new_merchant_category_i
    new_transactions_features = new_transactions_features.drop(columns = ['new_purchase_date_max', 'new_purchas
    new_transactions_features = reduce_mem_usage(new_transactions_features)

    agg_dict = {
    'purchase_amount': ['sum', 'mean', 'min', 'max', 'std']
            }
    category_col = ['category_1_0', 'category_1_1', 'category_2_1', 'category_2_2', 'category_2_3', 'category_2
                    'category_2_5', 'category_3_0', 'category_3_1', 'category_3_2']
    hist_category_features = category_aggregate_featurization(historical_transactions, category_col, groupby, a
                                                              prefix = 'hist')
    hist_category_features = reduce_mem_usage(hist_category_features)
    new_category_features = category_aggregate_featurization(new_transactions, category_col, groupby, agg_dict,
                                                             prefix = 'new')
    new_category_features = reduce_mem_usage(new_category_features)

    hist_month_lag_features = month_lag_aggregate_featurization(historical_transactions, groupby, agg_dict, pre
    hist_month_lag_features = reduce_mem_usage(hist_month_lag_features)
    new_month_lag_features = month_lag_aggregate_featurization(new_transactions, groupby, agg_dict)
    new_month_lag_features = reduce_mem_usage(new_month_lag_features)

    groupby1 = 'card_id'
    groupby2 = 'installments'
    columns = ['purchase_amount', 'authorized_flag']
    agg_dict = {
    'authorized_flag': ['sum', 'mean'],
    'purchase_amount': ['sum', 'mean', 'min', 'max', 'std']
    }
    hist_installments_features = successive_agg_featurization(historical_transactions, groupby1, groupby2, colu
                                                              prefix = 'hist')
    hist_installments_features = reduce_mem_usage(hist_installments_features)
    new_installments_features = successive_agg_featurization(new_transactions, groupby1, groupby2, columns, agg
                                                             prefix = 'new')
    new_installments_features = reduce_mem_usage(new_installments_features)

    quantiles = [0.012, 0.02, 0.05, 0.2, 0.5, 0.8, 0.96, 0.992, 1.0]
    hist_rfm_feature = rfm_feature(historical_transactions, quantiles)
    hist_rfm_feature = reduce_mem_usage(hist_rfm_feature)

    all_transaction_features = pd.merge(hist_transactions_features, new_transactions_features, on = 'card_id',
    all_transaction_features = pd.merge(all_transaction_features, hist_category_features, on = 'card_id', how =
    all_transaction_features = pd.merge(all_transaction_features, new_category_features, on = 'card_id', how =
    all_transaction_features = pd.merge(all_transaction_features, hist_month_lag_features, on = 'card_id', how
    all_transaction_features = pd.merge(all_transaction_features, new_month_lag_features, on = 'card_id', how =
    all_transaction_features = pd.merge(all_transaction_features, hist_installments_features, on = 'card_id', h
    all_transaction_features = pd.merge(all_transaction_features, new_installments_features, on = 'card_id', ho
    all_transaction_features = pd.merge(all_transaction_features, hist_rfm_feature, on = 'card_id', how = 'left
    all_transaction_features = reduce_mem_usage(all_transaction_features)
    all_transaction_features.to_csv('data/all_transaction_features.csv')

    del hist_transactions_features
    del new_transactions_features
    del hist_category_features
    del new_category_features
    del hist_month_lag_features
    del new_month_lag_features
    del hist_installments_features
    del new_installments_features
    del hist_rfm_feature
    print("Featurization of historical and new transactions Completed.........")

    return all_transaction_features
```

Final Data Preparation

In [18]:
```python
def data_prepare(train, test, all_transaction_features):
    """This function prepares the final train data with all features and returns the
    featurized train dataset. It takes train set and transaction features dataset as
    dataframe."""

    train = pd.merge(train, all_transaction_features, on = 'card_id', how = 'left')
    train.fillna(value = 0, inplace = True)
    test = pd.merge(test, all_transaction_features, on = 'card_id', how = 'left')
    test.fillna(value = 0, inplace = True)

    train.to_csv('data/final_train.csv', index = False)
    test.to_csv('data/final_test.csv', index = False)

    return train, test
```

Building Model

Function to build model.

In [19]:
```python
def build_model(train):
    """This function build models from the train set and return the trained models.
    It take featurized train dataframe as input."""

    print("Building the models.........")
    Y_train = train['target']
    Outlier = train['outlier']
    X_train = train.drop(columns = ['card_id', 'target', 'outlier'])

    parameters = {
        'objective'        : 'reg:squarederror',
        'learning_rate'    : 0.01,
        'eval_metric'      : 'rmse',
        'tree_method'      : 'gpu_hist',
        'predictor'        : 'gpu_predictor',
        'random_state'     : 9,
        'verbosity'        : 0,
        'max_depth'        : 7,
        'subsample'        : 0.7145610313690366,
        'colsample_bytree' : 0.364896100159906,
        'min_split_loss'   : 2.2685374838074592,
        'min_child_weight' : 16.579787389902428,
        'reg_alpha'        : 9.874511648120071,
        'reg_lambda'       : 3.474818860996104
    }

    folds = StratifiedKFold(n_splits = 4, shuffle = True, random_state = 9)
    for fold, (train_idx, val_idx) in enumerate(folds.split(X_train, Outlier.values)):
        train_data = xgb.DMatrix(X_train.iloc[train_idx], label = Y_train.iloc[train_idx])
        val_data = xgb.DMatrix(X_train.iloc[val_idx], label = Y_train.iloc[val_idx])
        reggressor_XGB = xgb.train(params = parameters, dtrain = train_data,
                                   evals = [(train_data, 'train'), (val_data, 'eval')], num_boost_round = 10000
                                   early_stopping_rounds = 500, verbose_eval = False)
        dump(reggressor_XGB, "".join(('data/Model', str(fold + 1), '.sav')))
    print("Buiding of models Completed.........")

    return
```

Function to predict target.

In [20]:
```python
def predict(X_test, model):
    '''This function predicts and returns the target value of test data.
    It takes X_test as dataframe and regressor model for input.'''

    Y_test_pred = model.predict(xgb.DMatrix(X_test), iteration_range = (0, model.best_iteration))

    return Y_test_pred[0]
```

Function 1

In [21]:
```python
def function_1(card_id):
    """This function include entire pipeline, from data preprocessing to making final predictions.
    It take in card_id from test as string for input and return loyalty score for the card_id."""

    if not os.path.exists("data/final_train.csv") or not os.path.exists("data/final_test.csv"):

        if not os.path.exists("data/train_featurized.csv") or not os.path.exists("data/test_featurized.csv") or
        not os.path.exists("data/all_transaction_features.csv"):

            if not os.path.exists("data/train_processed.csv") or not os.path.exists("data/test_processed.csv")
            not os.path.exists("data/historical_transactions_processed.csv") or\
            not os.path.exists("data/new_transactions_processed.csv"):

                train, test, historical_transactions, new_transactions = load_data()
                train, test = process_train_test(train, test)
                historical_transactions, new_transactions = process_transactions(historical_transactions, new_t
                train, test = feature_train_test(train, test)
                all_transaction_features = feature_transactions(historical_transactions, new_transactions)
                final_train, final_test = data_prepare(train, test, all_transaction_features)

            else:
                train = pd.read_csv("data/train_processed.csv")
                test = pd.read_csv("data/test_processed.csv")
                train, test = feature_train_test(train, test)
                historical_transactions = pd.read_csv("data/historical_transactions_processed.csv")
                new_transactions = pd.read_csv("data/new_transactions_processed.csv")
                all_transaction_features = feature_transactions(historical_transactions, new_transactions)
                final_train, final_test = data_prepare(train, test, all_transaction_features)

        else:
            train = pd.read_csv('data/train_featurized.csv')
            test = pd.read_csv('data/test_featurized.csv')
            all_transaction_features = pd.read_csv('data/all_transaction_features.csv')
            final_train, final_test = data_prepare(train, test, all_transaction_features)

    else:
        final_test = pd.read_csv('data/final_test.csv')
        final_test = reduce_mem_usage(final_test)

    if not os.path.exists("data/Model1.sav") or not os.path.exists("data/Model2.sav") or\
    not os.path.exists("data/Model3.sav") or not os.path.exists("data/Model4.sav"):
        final_train = pd.read_csv('data/final_train.csv')
        final_train = reduce_mem_usage(final_train)
        build_model(final_train)

    if str(card_id) in final_test['card_id'].astype('string').values:
        X_test = final_test[final_test['card_id'] == str(card_id)].drop(columns = ['card_id'])
        y = 0
        for i in range(4):
            model = load("".join(("data/Model", str(i + 1), ".sav")))
            y += (predict(X_test, model) / 4)
        return y
    else:
        print("Sorry, no data available for Card Id", card_id)
        return
```

Function 2

In [22]:
```python
def function_2(card_id, target):
    """This function include entire pipeline, from data preprocessing to making final predictions.
    It takes in card_id from test as string and loyalty score for input and returns the evaluation
    mertric for the model."""

    Y_test_pred = function_1(card_id)
    rmse = np.sqrt(mean_squared_error([target], [Y_test_pred]))

    return rmse
```

In [23]:
```python
card_id = "C_ID_0ab67a22ab"
test_pred = function_1(card_id)
print("The Loyalty Score for Card ID {} is {}".format(card_id, test_pred))
```

```
The Loyalty Score for Card ID C_ID_0ab67a22ab is -3.8906837105751038
```

In [24]:
```python
target = -3.8906837105751038
rsme = function_2(card_id, target)
print("The RSME Score for Model is", rsme)
```

```
The RSME Score for Model is 0.0
```