Elo Model Building and Training

This notebook focuses on building different regression models for the Elo Merchant Category Recommendation problem. We analyze the performance of different models and choose the best model for our final prediction.

In [1]:
```
import numpy as np
import pandas as pd
import warnings
warnings.filterwarnings('ignore')
import pickle
from prettytable import PrettyTable
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
import lightgbm as lgb
import xgboost as xgb
import optuna
```

Function to reduce the memory usage by any pandas dataframe variable

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

    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
```

Loading featurized train and test dataset

In [3]:
```
train = pd.read_csv('data/featurized_train.csv')
test = pd.read_csv('data/featurized_test.csv')
```

In [4]:
```
train = reduce_mem_usage(train)
test = reduce_mem_usage(test)
```

```
Mem. usage decreased to 176.00 Mb (67.3% reduction)
Mem. usage decreased to 113.53 Mb (65.3% reduction)
```

We will use the target column in train set as Y train and outlier column as Outlier array to stratify while splitting data. Then we will drop the card id, target and outlier from train and card id from test.

In [5]:
```
Y_train = train['target']
Outlier = train['outlier']
X_train = train.drop(columns = ['card_id', 'target', 'outlier'])
X_test = test.drop(columns = ['card_id'])
```

Model Training

We will be training different models for our problem and analyze which model performs better. There is large variation in values of different features since the values are not normalized, so we will be using non-linear regression models for our problem.

Baseline Model

The Baseline model will predict the mean of the target values of train as target for all test points. The RSME score of the Baseline model will provide us with benchmark from where we have to increase our model performance. Each of our model should have RMSE score less than the RSME score of the Baseline model.

In [6]:
```
Y_train_mean = pd.DataFrame(Y_train, columns = ['target'])
Y_train_mean['target'] = Y_train.astype('float').mean()
rsme_baseline = np.sqrt(mean_squared_error(Y_train, Y_train_mean))
print("RSME score for Baseline Model: ", rsme_baseline)
```

```
RSME score for Baseline Model:  3.850440680607971
```

KNNRegressor Model

First non-linear model we will use is KNN Regressor which is the simplest of the regression models. We will use Grid Search to find the optimum value of number of neighbors and then train the final model with that number of neighbors.

In [7]:
```
knn_model = KNeighborsRegressor(algorithm = 'kd_tree')
parameters = {"n_neighbors" : [1, 2, 5, 10, 50, 100, 200]}
knn_folds = StratifiedKFold(n_splits = 4, random_state = 9, shuffle = True).split(X_train, Outlier.values)
regressor_KNN = GridSearchCV(knn_model, parameters, cv = knn_folds, scoring = 'neg_mean_squared_error', n_jobs
```

In [8]:
```
regressor_KNN.fit(X_train, Y_train)
```

Out[8]:
```
GridSearchCV(cv=<generator object _BaseKFold.split at 0x000001CEBC88E3B0>,
             estimator=KNeighborsRegressor(algorithm='kd_tree'), n_jobs=-1,
             param_grid={'n_neighbors': [1, 2, 5, 10, 50, 100, 200]},
             scoring='neg_mean_squared_error')
```

In [9]:
```
best_params = regressor_KNN.best_params_
print("RSME :", np.sqrt(abs(regressor_KNN.best_score_)))
print("Best Hyperparameters")
print('-' * 20)
for hyperparameter, value in best_params.items():
    print(hyperparameter, ' : ', value)
```

```
RSME : 3.8250306371322047
Best Hyperparameters
--------------------
n_neighbors  :  200
```

We will now use the best parameters for KNN Regressor obtained from grid search to train the KNN Regressor Model and check its performance using RMSE score.

In [7]:
```
Y_train_pred = np.zeros(len(X_train))
knn_folds = StratifiedKFold(n_splits = 4, shuffle = True, random_state = 9)

for fold, (train_idx, val_idx) in enumerate(knn_folds.split(X_train, Outlier.values)):
    print("Training for fold {}.........".format(fold + 1))
    regressor_KNN = KNeighborsRegressor(n_neighbors = 200)
    regressor_KNN.fit(X_train.iloc[train_idx], Y_train.iloc[train_idx])
    Y_train_pred[val_idx] = regressor_KNN.predict(X_train.iloc[val_idx])
```

```
Training for fold 1.........
Training for fold 2.........
Training for fold 3.........
Training for fold 4.........
```

In [8]:
```
rsme_knn = np.sqrt(mean_squared_error(Y_train, Y_train_pred))
print("RSME score for KNN Regressor model: ", rsme_knn)
```

```
RSME score for KNN Regressor model:  3.824844001316022
```

The KNNRegressor model with RMSE score of 3.8248 shows a small improvement over the Baseline model. We will now build some complex models for our problem.

XGBoost Model

Now we will build a XGBoost model for our problem. But before we build the final XGBoost model we will use Optuna for finding the optimum hyperparameters for the model.

In [7]:
```
def objective(trial):
    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'        : trial.suggest_int('max_depth', 1, 8),
        'subsample'        : trial.suggest_uniform('subsample', 0.1, 1),
        'colsample_bytree' : trial.suggest_uniform('colsample_bytree ', 0.1, 1),
        'min_split_loss'   : trial.suggest_uniform('min_split_loss', 0, 10),
        'min_child_weight' : trial.suggest_uniform('min_child_weight', 0, 32),
        'reg_alpha'        : trial.suggest_uniform('reg_alpha', 0.1, 10),
        'reg_lambda'       : trial.suggest_uniform('reg_lambda', 0.1, 10)
            }

    Y_train_pred = np.zeros(len(X_train))
    xgb_folds = StratifiedKFold(n_splits = 4, shuffle = True, random_state = 9)

    for fold, (train_idx, val_idx) in enumerate(xgb_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)
        Y_train_pred[val_idx] = reggressor_XGB.predict(xgb.DMatrix(X_train.iloc[val_idx]),
                                                       iteration_range = (0, reggressor_XGB.best_iteration))

    return np.sqrt(mean_squared_error(Y_train, Y_train_pred))
```

In [8]:
```
study = optuna.create_study()
study.optimize(objective, n_trials = 20)
```

```
[I 2022-04-06 15:19:23,374] A new study created in memory with name: no-name-53d31acd-a906-4c5b-9c4a-2c7f6dad112d
[I 2022-04-06 15:27:57,693] Trial 0 finished with value: 3.6619383657128575 and parameters: {'max_depth': 4, 'subsample': 0.9284543408473314, 'colsample_bytree ': 0.6142413275865513, 'min_split_loss': 8.771818142117235, 'min_child_weight': 28.626850478354413, 'reg_alpha': 5.950270078225057, 'reg_lambda': 5.668986265787472}. Best is trial 0 with value: 3.6619383657128575.
[I 2022-04-06 15:35:32,200] Trial 1 finished with value: 3.6567083715942403 and parameters: {'max_depth': 6, 'subsample': 0.8479080538654751, 'colsample_bytree ': 0.7036818428596705, 'min_split_loss': 0.878213388299679, 'min_child_weight': 24.51261521795519, 'reg_alpha': 9.93772670077798, 'reg_lambda': 2.611761219922766}. Best is trial 1 with value: 3.6567083715942403.
[I 2022-04-06 15:43:34,915] Trial 2 finished with value: 3.661179407715765 and parameters: {'max_depth': 4, 'subsample': 0.9230392651263223, 'colsample_bytree ': 0.5954651886785793, 'min_split_loss': 0.0991246125709011, 'min_child_weight': 30.790700410321158, 'reg_alpha': 8.912795282576779, 'reg_lambda': 9.581735891069085}. Best is trial 1 with value: 3.6567083715942403.
[I 2022-04-06 15:55:16,528] Trial 3 finished with value: 3.69023803173204 and parameters: {'max_depth': 2, 'subsample': 0.33985985329657123, 'colsample_bytree ': 0.5530136153382356, 'min_split_loss': 8.336412225346555, 'min_child_weight': 3.42250016008364, 'reg_alpha': 1.8834159367371457, 'reg_lambda': 1.50146369534415}. Best is trial 1 with value: 3.6567083715942403.
[I 2022-04-06 16:07:52,306] Trial 4 finished with value: 3.6880772942702054 and parameters: {'max_depth': 2, 'subsample': 0.2911126243869341, 'colsample_bytree ': 0.8189063730329996, 'min_split_loss': 2.316506362362878, 'min_child_weight': 0.14967794537415813, 'reg_alpha': 7.959181116569342, 'reg_lambda': 7.8076422416540625}. Best is trial 1 with value: 3.6567083715942403.
[I 2022-04-06 16:19:38,272] Trial 5 finished with value: 3.6569816964289097 and parameters: {'max_depth': 8, 'subsample': 0.6244273905441523, 'colsample_bytree ': 0.7191547938903964, 'min_split_loss': 3.883828047862573, 'min_child_weight': 13.423328704687783, 'reg_alpha': 8.830768409869819, 'reg_lambda': 4.78882921075339}. Best is trial 1 with value: 3.6567083715942403.
[I 2022-04-06 16:25:33,059] Trial 6 finished with value: 3.6590863243347687 and parameters: {'max_depth': 5, 'subsample': 0.6152060398357531, 'colsample_bytree ': 0.43586852513673147, 'min_split_loss': 9.286939143166911, 'min_child_weight': 22.620957895542297, 'reg_alpha': 7.003260396141438, 'reg_lambda': 0.9892320211569767}. Best is trial 1 with value: 3.6567083715942403.
[I 2022-04-06 16:33:08,052] Trial 7 finished with value: 3.661364761454252 and parameters: {'max_depth': 5, 'subsample': 0.9662526242796118, 'colsample_bytree ': 0.9463173637783876, 'min_split_loss': 9.338805381090044, 'min_child_weight': 0.7464848479982074, 'reg_alpha': 1.6701445686696066, 'reg_lambda': 2.790671934700568}. Best is trial 1 with value: 3.6567083715942403.
[I 2022-04-06 16:43:21,758] Trial 8 finished with value: 3.6604296002061023 and parameters: {'max_depth': 8, 'subsample': 0.358509655175724, 'colsample_bytree ': 0.9958780665916114, 'min_split_loss': 6.939544709338007, 'min_child_weight': 30.279103429646913, 'reg_alpha': 2.84232042355615, 'reg_lambda': 0.3670849389279919}. Best is trial 1 with value: 3.6567083715942403.
[I 2022-04-06 16:52:16,042] Trial 9 finished with value: 3.6573103372949856 and parameters: {'max_depth': 7, 'subsample': 0.8443194357002407, 'colsample_bytree ': 0.9277765493557409, 'min_split_loss': 4.860710523470409, 'min_child_weight': 29.478152824142235, 'reg_alpha': 2.5614727782492954, 'reg_lambda': 5.0395706385219174}. Best is trial 1 with value: 3.6567083715942403.
[I 2022-04-06 16:58:33,723] Trial 10 finished with value: 3.6744161622040465 and parameters: {'max_depth': 6, 'subsample': 0.10990672992390699, 'colsample_bytree ': 0.12555869239676365, 'min_split_loss': 0.024190956410270914, 'min_child_weight': 16.791758181406195, 'reg_alpha': 4.630714944024064, 'reg_lambda': 3.0766975006421706}. Best is trial 1 with value: 3.6567083715942403.
[I 2022-04-06 17:10:33,363] Trial 11 finished with value: 3.6580256181442854 and parameters: {'max_depth': 8, 'subsample': 0.7117073467758669, 'colsample_bytree ': 0.7549007838853784, 'min_split_loss': 3.1819056678523827, 'min_child_weight': 10.320708628611172, 'reg_alpha': 9.956747439302987, 'reg_lambda': 4.922268935028661}. Best is trial 1 with value: 3.6567083715942403.
[I 2022-04-06 17:17:38,292] Trial 12 finished with value: 3.6543842961463846 and parameters: {'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}. Best is trial 12 with value: 3.6543842961463846.
[I 2022-04-06 17:23:50,396] Trial 13 finished with value: 3.655308624034834 and parameters: {'max_depth': 6, 'subsample': 0.7546894736891103, 'colsample_bytree ': 0.35953055604455697, 'min_split_loss': 1.7430072736882494, 'min_child_weight': 20.871581025950157, 'reg_alpha': 9.754453743092853, 'reg_lambda': 2.9917869001417916}. Best is trial 12 with value: 3.6543842961463846.
[I 2022-04-06 17:30:20,329] Trial 14 finished with value: 3.655754255676229 and parameters: {'max_depth': 6, 'subsample': 0.7378770130703372, 'colsample_bytree ': 0.338046217432016, 'min_split_loss': 1.9645302780920293, 'min_child_weight': 19.93997754531223, 'reg_alpha': 4.656227042466928, 'reg_lambda': 3.7495764104611213}. Best is trial 12 with value: 3.6543842961463846.
[I 2022-04-06 17:37:43,899] Trial 15 finished with value: 3.6559629537567555 and parameters: {'max_depth': 7, 'subsample': 0.5420137286855751, 'colsample_bytree ': 0.27513457857490214, 'min_split_loss': 6.1318482086460495, 'min_child_weight': 7.669472157495031, 'reg_alpha': 6.971738449754822, 'reg_lambda': 7.296388374083267}. Best is trial 12 with value: 3.6543842961463846.
[I 2022-04-06 17:52:00,139] Trial 16 finished with value: 3.6700415332758265 and parameters: {'max_depth': 3, 'subsample': 0.49550856948340094, 'colsample_bytree ': 0.40492581092133645, 'min_split_loss': 1.7655791763518962, 'min_child_weight': 16.698224788006556, 'reg_alpha': 0.4164559475927003, 'reg_lambda': 6.368655741414436}. Best is trial 12 with value: 3.6543842961463846.
[I 2022-04-06 18:06:45,351] Trial 17 finished with value: 3.6554290803978655 and parameters: {'max_depth': 7, 'subsample': 0.7352937090284963, 'colsample_bytree ': 0.18838194998649513, 'min_split_loss': 3.8813072182257575, 'min_child_weight': 20.728257731411414, 'reg_alpha': 8.663844153637069, 'reg_lambda': 1.894964817150832}. Best is trial 12 with value: 3.6543842961463846.
[I 2022-04-06 18:31:34,418] Trial 18 finished with value: 3.734257977121652 and parameters: {'max_depth': 1, 'subsample': 0.4306898046818125, 'colsample_bytree ': 0.46324135253079496, 'min_split_loss': 3.0587942845005154, 'min_child_weight': 25.125964232381428, 'reg_alpha': 7.383607233605659, 'reg_lambda': 3.7416948958534593}. Best is trial 12 with value: 3.6543842961463846.
[I 2022-04-06 18:37:46,675] Trial 19 finished with value: 3.6552227392232033 and parameters: {'max_depth': 6, 'subsample': 0.8036160540143876, 'colsample_bytree ': 0.3078380360844784, 'min_split_loss': 1.2862845938777738, 'min_child_weight': 13.362410445443684, 'reg_alpha': 5.934780234540893, 'reg_lambda': 3.967702509333143}. Best is trial 12 with value: 3.6543842961463846.
```

In [9]:
```
b_trial = study.best_trial
print('RSME : ', b_trial.value)
best_params = b_trial.params
print("Best Hyperparameters")
print('-' * 20)
for hyperparameter, value in best_params.items():
    print(hyperparameter, ' : ', value)
```

```
RSME :  3.6543842961463846
Best Hyperparameters
--------------------
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
```

In [10]:
```
with open('data/XGB_parameters', 'ab') as df_file:
    pickle.dump(best_params, df_file)
```

We will use the best hyperparameters obtained from optuna trial for building the final XGBoost model.

In [9]:
```
with open('data/XGB_parameters', 'rb') as df_file:
    best_params_xgb = pickle.load(df_file)
```

In [10]:
```
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'        : best_params_xgb.get('max_depth'),
    'subsample'        : best_params_xgb.get('subsample'),
    'colsample_bytree' : best_params_xgb.get('colsample_bytree'),
    'min_split_loss'   : best_params_xgb.get('min_split_loss'),
    'min_child_weight' : best_params_xgb.get('min_child_weight'),
    'reg_alpha'        : best_params_xgb.get('reg_alpha'),
    'reg_lambda'       : best_params_xgb.get('reg_lambda')
        }
```

In [11]:
```
Y_train_pred_xgb = np.zeros(len(X_train))
xgb_folds = StratifiedKFold(n_splits = 4, shuffle = True, random_state = 9)

for fold, (train_idx, val_idx) in enumerate(xgb_folds.split(X_train, Outlier.values)):
    print("Training for fold {}.........".format(fold + 1))
    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_d
                               num_boost_round = 10000, early_stopping_rounds = 500, verbose_eval = 1000)
    Y_train_pred_xgb[val_idx] = reggressor_XGB.predict(xgb.DMatrix(X_train.iloc[val_idx]),
                                                       iteration_range = (0, reggressor_XGB.best_iteration))
```

```
Training for fold 1.........
[0]     train-rmse:3.98423      eval-rmse:3.83439
[1000]  train-rmse:3.26844      eval-rmse:3.53679
[1369]  train-rmse:3.16492      eval-rmse:3.53823
Training for fold 2.........
[0]     train-rmse:3.93100      eval-rmse:3.99622
[1000]  train-rmse:3.21182      eval-rmse:3.72947
[1268]  train-rmse:3.13924      eval-rmse:3.73045
Training for fold 3.........
[0]     train-rmse:3.93589      eval-rmse:3.98138
[1000]  train-rmse:3.21848      eval-rmse:3.69194
[1355]  train-rmse:3.12377      eval-rmse:3.69378
Training for fold 4.........
[0]     train-rmse:3.93726      eval-rmse:3.97762
[1000]  train-rmse:3.22834      eval-rmse:3.67231
[1589]  train-rmse:3.07012      eval-rmse:3.67334
```

In [12]:
```
rsme_xgb = np.sqrt(mean_squared_error(Y_train, Y_train_pred_xgb))
print("RSME score for XGBoost model: ", rsme_xgb)
```

```
RSME score for XGBoost model:  3.6579771246304467
```

The XGBoost model with RSME score of 3.6579 has much better performance than the KNNRegressor and shows big improvement in RSME score from the Baseline model.

LightGBM Model

We will also build a LightGBM model for the problem to analyze if it performs better than the XGBoost model. Similar to XGBoost, we will use Optuna to find the best hyperparameters before building the final LightGBM model.

In [13]:
```
def objective(trial):
    parameters = {
        'objective'        : 'regression',
        'metric'           : 'rmse',
        'boosting_type'    : 'gbdt',
        'learning_rate'    : 0.01,
        'device'           : 'cpu',
        'n_jobs'           : -1,
        'verbosity'        : -1,
        'random_state'     : 9,
        'bagging_freq'     : 1,
        'bagging_seed'     : 9,
        'max_depth'        : trial.suggest_int('max_depth', 1, 16),
        'num_leaves'       : trial.suggest_int('num_leaves', 16, 128),
        'min_data_in_leaf' : trial.suggest_int('min_data_in_leaf', 8, 64),
        'min_child_weight' : trial.suggest_uniform('min_child_weight', 0, 32),
        'feature_fraction' : trial.suggest_uniform('feature_fraction', 0.1, 1.0),
        'bagging_fraction' : trial.suggest_uniform('bagging_fraction', 0.1, 1.0),
        'min_split_gain'   : trial.suggest_uniform('min_split_gain', 0, 10),
        'reg_alpha'        : trial.suggest_uniform('reg_alpha', 0, 10),
        'reg_lambda'       : trial.suggest_uniform('reg_lambda', 0, 10)
            }

    Y_train_pred = np.zeros(len(X_train))
    lgb_folds = StratifiedKFold(n_splits = 4, shuffle = True, random_state = 9)

    for fold, (train_idx, val_idx) in enumerate(lgb_folds.split(X_train, Outlier.values)):
        train_data = lgb.Dataset(X_train.iloc[train_idx], label = Y_train.iloc[train_idx])
        val_data = lgb.Dataset(X_train.iloc[val_idx], label = Y_train.iloc[val_idx])
        reggressor_LGB = lgb.train(params = parameters, train_set = train_data, valid_sets = [train_data, val_d
                                   num_boost_round = 10000, early_stopping_rounds = 500, verbose_eval = False)
        Y_train_pred[val_idx] = reggressor_LGB.predict(X_train.iloc[val_idx], num_iteration = reggressor_LGB.be

    return np.sqrt(mean_squared_error(Y_train, Y_train_pred))
```

In [14]:
```
study = optuna.create_study()
study.optimize(objective, n_trials = 20)
```

```
[I 2022-04-07 00:47:47,852] A new study created in memory with name: no-name-9168335d-dbc9-496e-8895-179b48181566
[I 2022-04-07 00:54:35,935] Trial 0 finished with value: 3.6630423850962637 and parameters: {'max_depth': 12, 'num_leaves': 71, 'min_data_in_leaf': 22, 'min_child_weight': 21.669760862270813, 'feature_fraction': 0.5991532629972278, 'bagging_fraction': 0.40395548684511795, 'min_split_gain': 4.348054423432296, 'reg_alpha': 2.217543805827545, 'reg_lambda': 3.039140014881019}. Best is trial 0 with value: 3.6630423850962637.
[I 2022-04-07 01:04:50,792] Trial 1 finished with value: 3.6574406809362014 and parameters: {'max_depth': 9, 'num_leaves': 42, 'min_data_in_leaf': 53, 'min_child_weight': 26.058615434144063, 'feature_fraction': 0.82205270610647, 'bagging_fraction': 0.8922895465650358, 'min_split_gain': 3.768155610827323, 'reg_alpha': 9.119062148441706, 'reg_lambda': 6.553709195366454}. Best is trial 1 with value: 3.6574406809362014.
[I 2022-04-07 01:26:40,094] Trial 2 finished with value: 3.687509677709659 and parameters: {'max_depth': 2, 'num_leaves': 53, 'min_data_in_leaf': 20, 'min_child_weight': 31.3497565196691, 'feature_fraction': 0.9412763383404444, 'bagging_fraction': 0.7882988513258217, 'min_split_gain': 2.9882013397167073, 'reg_alpha': 2.6681740629419326, 'reg_lambda': 0.12642142265436251}. Best is trial 1 with value: 3.6574406809362014.
[I 2022-04-07 01:31:50,941] Trial 3 finished with value: 3.6617145946640917 and parameters: {'max_depth': 8, 'num_leaves': 23, 'min_data_in_leaf': 34, 'min_child_weight': 6.0921731789187135, 'feature_fraction': 0.4807373630519206, 'bagging_fraction': 0.4318080651273023, 'min_split_gain': 6.19801713860399, 'reg_alpha': 1.2939084722117233, 'reg_lambda': 2.336503781610312}. Best is trial 1 with value: 3.6574406809362014.
[I 2022-04-07 01:42:10,222] Trial 4 finished with value: 3.6625471833940697 and parameters: {'max_depth': 4, 'num_leaves': 77, 'min_data_in_leaf': 9, 'min_child_weight': 21.76258486439517, 'feature_fraction': 0.5601161818456099, 'bagging_fraction': 0.9464732131946071, 'min_split_gain': 3.9458547409624725, 'reg_alpha': 5.818272316383561, 'reg_lambda': 8.874537586433622}. Best is trial 1 with value: 3.6574406809362014.
[I 2022-04-07 01:49:15,208] Trial 5 finished with value: 3.6663725637949165 and parameters: {'max_depth': 16, 'num_leaves': 120, 'min_data_in_leaf': 58, 'min_child_weight': 2.144902755552689, 'feature_fraction': 0.15557404142879028, 'bagging_fraction': 0.29981072618721505, 'min_split_gain': 3.3275569173944763, 'reg_alpha': 5.286998887674157, 'reg_lambda': 4.709183214556333}. Best is trial 1 with value: 3.6574406809362014.
[I 2022-04-07 01:58:06,639] Trial 6 finished with value: 3.6704289050163696 and parameters: {'max_depth': 5, 'num_leaves': 120, 'min_data_in_leaf': 36, 'min_child_weight': 11.679726089495535, 'feature_fraction': 0.15914698918145215, 'bagging_fraction': 0.14385242383875402, 'min_split_gain': 5.953101021397269, 'reg_alpha': 6.934271364173691, 'reg_lambda': 5.840012476765176}. Best is trial 1 with value: 3.6574406809362014.
[I 2022-04-07 02:05:54,039] Trial 7 finished with value: 3.671626781391197 and parameters: {'max_depth': 14, 'num_leaves': 94, 'min_data_in_leaf': 47, 'min_child_weight': 23.791540519168898, 'feature_fraction': 0.8570902070148612, 'bagging_fraction': 0.12859621679162855, 'min_split_gain': 1.1542536990525287, 'reg_alpha': 4.09914205683505, 'reg_lambda': 5.411445316124768}. Best is trial 1 with value: 3.6574406809362014.
[I 2022-04-07 02:18:34,728] Trial 8 finished with value: 3.737315632837233 and parameters: {'max_depth': 1, 'num_leaves': 80, 'min_data_in_leaf': 19, 'min_child_weight': 18.12503015676918, 'feature_fraction': 0.4386810790201414, 'bagging_fraction': 0.3559475215161121, 'min_split_gain': 5.1298000392298775, 'reg_alpha': 9.708434157961342, 'reg_lambda': 8.159726750172482}. Best is trial 1 with value: 3.6574406809362014.
[I 2022-04-07 02:28:15,670] Trial 9 finished with value: 3.6557832249966506 and parameters: {'max_depth': 8, 'num_leaves': 128, 'min_data_in_leaf': 28, 'min_child_weight': 18.712065960638746, 'feature_fraction': 0.5688613200905298, 'bagging_fraction': 0.7775705635193393, 'min_split_gain': 1.348505568825985, 'reg_alpha': 7.4747963808521245, 'reg_lambda': 3.597114670480944}. Best is trial 9 with value: 3.6557832249966506.
[I 2022-04-07 02:36:10,408] Trial 10 finished with value: 3.658582587846033 and parameters: {'max_depth': 9, 'num_leaves': 106, 'min_data_in_leaf': 43, 'min_child_weight': 12.7634358361123, 'feature_fraction': 0.6793235328878287, 'bagging_fraction': 0.6715660608010557, 'min_split_gain': 9.465408406844215, 'reg_alpha': 7.707641422798685, 'reg_lambda': 0.06895362504385805}. Best is trial 9 with value: 3.6557832249966506.
[I 2022-04-07 02:45:49,664] Trial 11 finished with value: 3.657962875779366 and parameters: {'max_depth': 9, 'num_leaves': 37, 'min_data_in_leaf': 47, 'min_child_weight': 29.73267330437201, 'feature_fraction': 0.7700104545097888, 'bagging_fraction': 0.9990589411241023, 'min_split_gain': 0.39769147902561297, 'reg_alpha': 9.984417672011926, 'reg_lambda': 7.129707869325249}. Best is trial 9 with value: 3.6557832249966506.
[I 2022-04-07 02:56:14,514] Trial 12 finished with value: 3.6560084785763785 and parameters: {'max_depth': 6, 'num_leaves': 52, 'min_data_in_leaf': 64, 'min_child_weight': 26.167439606538956, 'feature_fraction': 0.36887573276598606, 'bagging_fraction': 0.7975711598497612, 'min_split_gain': 1.8368667966461514, 'reg_alpha': 8.18813552385062, 'reg_lambda': 3.4181327302114215}. Best is trial 9 with value: 3.6557832249966506.
[I 2022-04-07 03:02:02,355] Trial 13 finished with value: 3.6558622632283373 and parameters: {'max_depth': 6, 'num_leaves': 60, 'min_data_in_leaf': 64, 'min_child_weight': 16.499103534101472, 'feature_fraction': 0.393610719209942, 'bagging_fraction': 0.6475390952499872, 'min_split_gain': 1.6693583942874644, 'reg_alpha': 7.922950190857678, 'reg_lambda': 3.360524687295772}. Best is trial 9 with value: 3.6557832249966506.
[I 2022-04-07 03:11:58,696] Trial 14 finished with value: 3.659372400334074 and parameters: {'max_depth': 12, 'num_leaves': 62, 'min_data_in_leaf': 29, 'min_child_weight': 17.087599310358023, 'feature_fraction': 0.3083320076825721, 'bagging_fraction': 0.6368495205157543, 'min_split_gain': 0.04083347943075921, 'reg_alpha': 7.013368742462264, 'reg_lambda': 1.6512446030953574}. Best is trial 9 with value: 3.6557832249966506.
[I 2022-04-07 03:21:26,572] Trial 15 finished with value: 3.658863713345061 and parameters: {'max_depth': 7, 'num_leaves': 95, 'min_data_in_leaf': 42, 'min_child_weight': 11.529503798106507, 'feature_fraction': 0.2786979520577261, 'bagging_fraction': 0.5576345885858514, 'min_split_gain': 2.0598778444750665, 'reg_alpha': 6.126715632617882, 'reg_lambda': 4.3855034044905485}. Best is trial 9 with value: 3.6557832249966506.
[I 2022-04-07 03:31:40,817] Trial 16 finished with value: 3.670509939142782 and parameters: {'max_depth': 3, 'num_leaves': 22, 'min_data_in_leaf': 28, 'min_child_weight': 14.75491911647246, 'feature_fraction': 0.6814024308355047, 'bagging_fraction': 0.7533163414910654, 'min_split_gain': 8.576141864288697, 'reg_alpha': 8.44372803353525, 'reg_lambda': 3.8264756089796728}. Best is trial 9 with value: 3.6557832249966506.
[I 2022-04-07 03:39:06,274] Trial 17 finished with value: 3.6631446808868633 and parameters: {'max_depth': 11, 'num_leaves': 90, 'min_data_in_leaf': 14, 'min_child_weight': 7.8857898799835, 'feature_fraction': 0.4574151852978893, 'bagging_fraction': 0.5172766626730054, 'min_split_gain': 2.0791580181573965, 'reg_alpha': 6.84804813238946, 'reg_lambda': 1.8005027088613534}. Best is trial 9 with value: 3.6557832249966506.
[I 2022-04-07 03:46:34,076] Trial 18 finished with value: 3.6550641942500763 and parameters: {'max_depth': 6, 'num_leaves': 108, 'min_data_in_leaf': 64, 'min_child_weight': 19.542744175146908, 'feature_fraction': 0.6463694894206278, 'bagging_fraction': 0.6959253014980463, 'min_split_gain': 7.309719954929109, 'reg_alpha': 4.6100383504763505, 'reg_lambda': 9.827071071570177}. Best is trial 18 with value: 3.6550641942500763.
[I 2022-04-07 03:57:33,329] Trial 19 finished with value: 3.6570494402472997 and parameters: {'max_depth': 11, 'num_leaves': 125, 'min_data_in_leaf': 28, 'min_child_weight': 19.72828031110244, 'feature_fraction': 0.6562130735667374, 'bagging_fraction': 0.8581991424448983, 'min_split_gain': 7.128826954687689, 'reg_alpha': 4.171482230038237, 'reg_lambda': 9.206931485860473}. Best is trial 18 with value: 3.6550641942500763.
```

In [19]:
```
b_trial = study.best_trial
print('RSME : ', b_trial.value)
best_params = b_trial.params
print("Best Hyperparameters")
print('-' * 20)
for hyperparameter, value in best_params.items():
    print(hyperparameter, ' : ', value)
```

```
RSME :  3.6550641942500763
Best Hyperparameters
--------------------
max_depth  :  6
num_leaves  :  108
min_data_in_leaf  :  64
min_child_weight  :  19.542744175146908
feature_fraction  :  0.6463694894206278
bagging_fraction  :  0.6959253014980463
min_split_gain  :  7.309719954929109
reg_alpha  :  4.6100383504763505
reg_lambda  :  9.827071071570177
```

In [15]:
```
with open('data/LGB_parameters', 'ab') as df_file:
    pickle.dump(best_params, df_file)
```

Now the best hyperparameters obtained from Optuna will be used to build the final LightGBM model.

In [13]:
```
with open('data/LGB_parameters', 'rb') as df_file:
    best_params_lgb = pickle.load(df_file)
```

In [14]:
```
parameters = {
    'objective'        : 'regression',
    'metric'           : 'rmse',
    'boosting_type'    : 'gbdt',
    'learning_rate'    : 0.01,
    'device'           : 'cpu',
    'n_jobs'           : -1,
    'verbosity'        : -1,
    'random_state'     : 9,
    'bagging_freq'     : 1,
    'bagging_seed'     : 9,
    'max_depth'        : best_params_lgb.get('max_depth'),
    'num_leaves'       : best_params_lgb.get('num_leaves'),
    'min_data_in_leaf' : best_params_lgb.get('min_data_in_leaf'),
    'min_child_weight' : best_params_lgb.get('min_child_weight'),
    'feature_fraction' : best_params_lgb.get('feature_fraction'),
    'bagging_fraction' : best_params_lgb.get('bagging_fraction'),
    'min_split_gain'   : best_params_lgb.get('min_split_gain'),
    'reg_alpha'        : best_params_lgb.get('reg_alpha'),
    'reg_lambda'       : best_params_lgb.get('reg_lambda')
        }
```

In [15]:
```
Y_train_pred_lgb = np.zeros(len(X_train))
lgb_folds = StratifiedKFold(n_splits = 4, shuffle = True, random_state = 9)

for fold, (train_idx, val_idx) in enumerate(lgb_folds.split(X_train, Outlier.values)):
    print("Training for fold {}.........".format(fold + 1))
    train_data = lgb.Dataset(X_train.iloc[train_idx], label = Y_train.iloc[train_idx])
    val_data = lgb.Dataset(X_train.iloc[val_idx], label = Y_train.iloc[val_idx])
    reggressor_LGB = lgb.train(params = parameters, train_set = train_data, valid_sets = [train_data, val_data]
                               num_boost_round = 10000, verbose_eval = 1000, early_stopping_rounds = 500)
    Y_train_pred_lgb[val_idx] = reggressor_LGB.predict(X_train.iloc[val_idx], num_iteration = reggressor_LGB.be
```

```
Training for fold 1.........
Training until validation scores don't improve for 500 rounds
[1000]  training's rmse: 3.51008        valid_1's rmse: 3.53725
[2000]  training's rmse: 3.39051        valid_1's rmse: 3.53775
Early stopping, best iteration is:
[1608]  training's rmse: 3.435  valid_1's rmse: 3.53617
Training for fold 2.........
Training until validation scores don't improve for 500 rounds
[1000]  training's rmse: 3.44442        valid_1's rmse: 3.72588
Early stopping, best iteration is:
[1038]  training's rmse: 3.43916        valid_1's rmse: 3.72561
Training for fold 3.........
Training until validation scores don't improve for 500 rounds
[1000]  training's rmse: 3.45567        valid_1's rmse: 3.68775
Early stopping, best iteration is:
[1229]  training's rmse: 3.42736        valid_1's rmse: 3.68719
Training for fold 4.........
Training until validation scores don't improve for 500 rounds
[1000]  training's rmse: 3.46697        valid_1's rmse: 3.67038
[2000]  training's rmse: 3.35138        valid_1's rmse: 3.66958
Early stopping, best iteration is:
[1514]  training's rmse: 3.40552        valid_1's rmse: 3.66851
```

In [16]:
```
rsme_lgb = np.sqrt(mean_squared_error(Y_train, Y_train_pred_lgb))
print("RSME score for LGBM model: ", rsme_lgb)
```

```
RSME score for LGBM model:  3.6550641942500763
```

The LightGBM model with RSME score of 3.6551 is the best of all the models with XGBoost not far behind. To improve the RSME score even more, we will use stacked models of LightGBM and XGBoost with different weightage.

Stacked Model

We will use different weightage for the XGBoost and LightGBM models to decide the weightage for our final Stacked model.

In [17]:
```
for i in range(10,100,10):
    Y_train_pred_stack = ((i / 100) * Y_train_pred_xgb) + (((100 - i) / 100) * Y_train_pred_lgb)
    print("RSME score for Stacked Model with {}% weightage to XGBoost and {}% weightage to LightGBM : {}".forma
```

```
RSME score for Stacked Model with 10% weightage to XGBoost and 90% weightage to LightGBM : 3.654806730014835
RSME score for Stacked Model with 20% weightage to XGBoost and 80% weightage to LightGBM : 3.6546712326313924
RSME score for Stacked Model with 30% weightage to XGBoost and 70% weightage to LightGBM : 3.6546577156656053
RSME score for Stacked Model with 40% weightage to XGBoost and 60% weightage to LightGBM : 3.6547661804708893
RSME score for Stacked Model with 50% weightage to XGBoost and 50% weightage to LightGBM : 3.6549966161875393
RSME score for Stacked Model with 60% weightage to XGBoost and 40% weightage to LightGBM : 3.6553489997481643
RSME score for Stacked Model with 70% weightage to XGBoost and 30% weightage to LightGBM : 3.655823295889232
RSME score for Stacked Model with 80% weightage to XGBoost and 20% weightage to LightGBM : 3.656419457168701
RSME score for Stacked Model with 90% weightage to XGBoost and 10% weightage to LightGBM : 3.657137423989734
```

In [26]:
```
rsme_stack = np.sqrt(mean_squared_error(Y_train, ((0.3 * Y_train_pred_xgb) + (0.7 * Y_train_pred_lgb))))
print("RSME score for Stacked model: ", rsme_stack)
```

```
RSME score for Stacked model:  3.6546577156656053
```

The Stacked Model with 30% weightage to XGBoost and 70% weightage to LightGBM provides a small improvement in RSME score from our previous best of LightGBM.

Stacked Model using Meta Learner

Now we will build another Stacked model but with a meta learner and see if it is better than the rest of our models. We will be using Ridge Regressor with predictions of XGBoost and LightGBM as input and the Y train as target values.

In [19]:
```
meta_train = np.vstack([Y_train_pred_xgb, Y_train_pred_lgb]).transpose()
```

In [20]:
```
meta_model = Ridge()
parameters = {"alpha" : [0.0001, 0.001, 0.01,0.1, 1.0]}
meta_folds = StratifiedKFold(n_splits = 3, random_state = 9, shuffle = True).split(meta_train, Outlier.values)
regressor_meta = GridSearchCV(meta_model, parameters, cv = meta_folds, scoring = 'neg_mean_squared_error',)
```

In [21]:
```
regressor_meta.fit(meta_train, Y_train)
```

Out[21]:
```
GridSearchCV(cv=<generator object _BaseKFold.split at 0x000001F5627BB300>,
             estimator=Ridge(),
             param_grid={'alpha': [0.0001, 0.001, 0.01, 0.1, 1.0]},
             scoring='neg_mean_squared_error')
```

In [22]:
```
best_params = regressor_meta.best_params_
print("Best Hyperparameters")
print('-' * 20)
for hyperparameter, value in best_params.items():
    print(hyperparameter, ' : ', value)
```

```
Best Hyperparameters
--------------------
alpha  :  1.0
```

In [24]:
```
Y_train_pred_meta = np.zeros(len(meta_train))
meta_folds = StratifiedKFold(n_splits = 4, random_state = 9, shuffle = True)

for fold, (train_idx, val_idx) in enumerate(meta_folds.split(meta_train, Outlier.values)):
    print("Training for fold {}.........".format(fold + 1))
    regressor_meta = Ridge(alpha = 1.0)
    regressor_meta.fit(meta_train[train_idx], Y_train.iloc[train_idx].values)
    Y_train_pred_meta[val_idx] = regressor_meta.predict(meta_train[val_idx])
```

```
Training for fold 1.........
Training for fold 2.........
Training for fold 3.........
Training for fold 4.........
```

In [25]:
```
rsme_meta_stack = np.sqrt(mean_squared_error(Y_train, Y_train_pred_meta))
print("RSME score for Stacked model with Meta Learner: ", rsme_meta_stack)
```

```
RSME score for Stacked model with Meta Learner:  3.654871417535515
```

The Meta Leaner Stacked model with RSME score of 3.6548 has performed little better than LightGBM but poorly compared to Weighted Stacked models.

Final RSME Scores of all Models

In [27]:
```
myTable = PrettyTable(["Model", "RSME Score"])
myTable.add_row(["Baseline", rsme_baseline])
myTable.add_row(["KNNRegressor", rsme_knn])
myTable.add_row(["XGBoost", rsme_xgb])
myTable.add_row(["LightGBM", rsme_lgb])
myTable.add_row(["Simple Stacked", rsme_stack])
myTable.add_row(["Meta-learner Stacked", rsme_meta_stack])
print(myTable)
```

```
+----------------------+--------------------+
|        Model         |     RSME Score     |
+----------------------+--------------------+
|       Baseline       | 3.850440680607971  |
|     KNNRegressor     | 3.824844001316022  |
|       XGBoost        | 3.6579771246304467 |
|       LightGBM       | 3.6550641942500763 |
|    Simple Stacked    | 3.6546577156656053 |
| Meta-learner Stacked | 3.654871417535515  |
+----------------------+--------------------+
```

Submission

In [28]:
```
with open('data/XGB_parameters', 'rb') as df_file:
    best_params_xgb = pickle.load(df_file)
```

In [29]:
```
with open('data/LGB_parameters', 'rb') as df_file:
    best_params_lgb = pickle.load(df_file)
```

In [30]:
```
def predict(X_train, Y_train, X_test, Outlier, param, num_splits = 4, num_round = 10000, model = 'lgb'):
    '''This function predicts and returns the target value of test data by training model on
    train data. It takes X_train, X_test and Y_train as dataframe, parameters for model as dictionary,
    number of boosting rounds for model and whether to train LightGBM or XGBoost model as string.'''
    Y_test_pred = np.zeros(len(X_test))
    folds = StratifiedKFold(n_splits = num_splits, shuffle = True, random_state = 9)

    if model == 'lgb':

        parameters = {
            'objective'        : 'regression',
            'metric'           : 'rmse',
            'boosting_type'    : 'gbdt',
            'learning_rate'    : 0.01,
            'device'           : 'cpu',
            'n_jobs'           : -1,
            'verbose'          : -1,
            'random_state'     : 9,
            'bagging_freq'     : 1,
            'bagging_seed'     : 9,
            'max_depth'        : param.get('max_depth'),
            'num_leaves'       : param.get('num_leaves'),
            'min_data_in_leaf' : param.get('min_data_in_leaf'),
            'min_child_weight' : param.get('min_child_weight'),
            'feature_fraction' : param.get('feature_fraction'),
            'bagging_fraction' : param.get('bagging_fraction'),
            'min_split_gain'   : param.get('min_split_gain'),
            'reg_alpha'        : param.get('reg_alpha'),
            'reg_lambda'       : param.get('reg_lambda')
            }

        for fold, (train_idx, val_idx) in enumerate(folds.split(X_train, Outlier.values)):
            train_data = lgb.Dataset(X_train.iloc[train_idx], label = Y_train.iloc[train_idx])
            val_data = lgb.Dataset(X_train.iloc[val_idx], label = Y_train.iloc[val_idx])
            reggressor_LGB = lgb.train(params = parameters, train_set = train_data, valid_sets = [train_data, v
                                       num_boost_round = num_round, early_stopping_rounds = 500, verbose_eval = Fa
            Y_test_pred += (reggressor_LGB.predict(X_test, num_iteration = reggressor_LGB.best_iteration) / num_

        return Y_test_pred

    elif model == 'xgb':

        parameters = {
            'objective'        : 'reg:squarederror',
            'learning_rate'    : 0.01,
            'eval_metric'      : 'rmse',
            'tree_method'      : 'gpu_hist',
            'predictor'        : 'gpu_predictor',
            'random_state'     : 9,
            'max_depth'        : param.get('max_depth'),
            'subsample'        : param.get('subsample'),
            'colsample_bytree' : param.get('colsample_bytree'),
            'min_split_loss'   : param.get('min_split_loss'),
            'min_child_weight' : param.get('min_child_weight'),
            'reg_alpha'        : param.get('reg_alpha'),
            'reg_lambda'       : param.get('reg_lambda')
        }

        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 = n
                                       early_stopping_rounds = 500, verbose_eval = False)
            Y_test_pred += (reggressor_XGB.predict(xgb.DMatrix(X_test),
                                                   iteration_range = (0, reggressor_XGB.best_iteration)) / num_

        return Y_test_pred
```

In [31]:
```
Y_test_pred_xgb = predict(X_train, Y_train, X_test, Outlier, best_params_xgb, model = 'xgb')
```

In [32]:
```
sub_xgb = pd.DataFrame({"card_id":test["card_id"].values})
sub_xgb["target"] = Y_test_pred_xgb
sub_xgb.to_csv("data/submit_xgb.csv", index = False)
```

XGBoost Model Kaggle Score

In [33]:
```
Y_test_pred_lgb = predict(X_train, Y_train, X_test, Outlier, best_params_lgb, model = 'lgb')
```

In [34]:
```
sub_lgb = pd.DataFrame({"card_id":test["card_id"].values})
sub_lgb["target"] = Y_test_pred_lgb
sub_lgb.to_csv("data/submit_lgb.csv", index = False)
```

LightGBM Model Kaggle Score

In [35]:
```
Y_test_pred_stack = (0.1 * Y_test_pred_xgb) + (0.9 * Y_test_pred_lgb)
```

In [36]:
```
sub_stack = pd.DataFrame({"card_id":test["card_id"].values})
sub_stack["target"] = Y_test_pred_stack
sub_stack.to_csv("data/submit_stack.csv", index = False)
```

Stacked Model Kaggle Score

In [37]:
```
meta_train = np.vstack([Y_train_pred_xgb, Y_train_pred_lgb]).transpose()
meta_test = np.vstack([Y_test_pred_xgb, Y_test_pred_lgb]).transpose()
```

In [40]:
```
Y_train_pred_meta = np.zeros(len(meta_train))
Y_test_pred_meta = np.zeros(len(meta_test))
meta_folds = StratifiedKFold(n_splits = 4, random_state = 9, shuffle = True)

for fold, (train_idx, val_idx) in enumerate(meta_folds.split(meta_train, Outlier.values)):
    regressor_meta = Ridge(alpha = 1.0)
    regressor_meta.fit(meta_train[train_idx], Y_train.iloc[train_idx].values)
    Y_train_pred_meta[val_idx] = regressor_meta.predict(meta_train[val_idx])
    Y_test_pred_meta += (regressor_meta.predict(meta_test) / meta_folds.n_splits)
```

In [41]:
```
sub_meta = pd.DataFrame({"card_id":test["card_id"].values})
sub_meta["target"] = Y_test_pred_meta
sub_meta.to_csv("data/submit_meta.csv", index = False)
```

Meta Learner Stacked Model Kaggle Score