lightgbm-GDBT

July 13, 2019

```
In [11]: import numpy as np
         import pandas as pd
         from sklearn import preprocessing
         import gc, sys
         gc.enable()

         import lightgbm as lgb
```

```
In [6]: def feature_engineering(is_train=True,debug=True):
            test_idx = None
            if is_train:
                print("processing train.csv")
                if debug == True:
                    df = pd.read_csv('./train_V2.csv', nrows=10000)
                else:
                    df = pd.read_csv('./train_V2.csv')

                df = df[df['maxPlace'] > 1]
            else:
                print("processing test.csv")
                df = pd.read_csv('./test_V2.csv')
                test_idx = df.Id

            print("remove some columns")
            target = 'winPlacePerc'

            print("Adding Features")

            df['headshotrate'] = df['kills']/df['headshotKills']
            df['killStreakrate'] = df['killStreaks']/df['kills']
            df['healthitems'] = df['heals'] + df['boosts']
            df['totalDistance'] = df['rideDistance'] + df["walkDistance"] + df["swimDistance"]
            df['killPlace_over_maxPlace'] = df['killPlace'] / df['maxPlace']
            df['headshotKills_over_kills'] = df['headshotKills'] / df['kills']
            df['distance_over_weapons'] = df['totalDistance'] / df['weaponsAcquired']
            df['walkDistance_over_heals'] = df['walkDistance'] / df['heals']
            df['walkDistance_over_kills'] = df['walkDistance'] / df['kills']
```

```
df['killsPerWalkDistance'] = df['kills'] / df['walkDistance']
df["skill"] = df["headshotKills"] + df["roadKills"]

df[df == np.Inf] = np.NaN
df[df == np.NINF] = np.NaN

print("Removing Na's From DF")
df.fillna(0, inplace=True)


features = list(df.columns)
features.remove("Id")
features.remove("matchId")
features.remove("groupId")
features.remove("matchType")

# matchType = pd.get_dummies(df['matchType'])
# df = df.join(matchType)

y = None


if is_train:
    print("get target")
    y = np.array(df.groupby(['matchId','groupId'])[target].agg('mean'), dtype=np.flo
    features.remove(target)

print("get group mean feature")
agg = df.groupby(['matchId','groupId'])[features].agg('mean')
agg_rank = agg.groupby('matchId')[features].rank(pct=True).reset_index()

if is_train: df_out = agg.reset_index()[['matchId','groupId']]
else: df_out = df[['matchId','groupId']]

df_out = df_out.merge(agg.reset_index(), suffixes=["", ""], how='left', on=['matchId
df_out = df_out.merge(agg_rank, suffixes=["_mean", "_mean_rank"], how='left', on=['m

print("get group max feature")
agg = df.groupby(['matchId','groupId'])[features].agg('max')
agg_rank = agg.groupby('matchId')[features].rank(pct=True).reset_index()
df_out = df_out.merge(agg.reset_index(), suffixes=["", ""], how='left', on=['matchId
df_out = df_out.merge(agg_rank, suffixes=["_max", "_max_rank"], how='left', on=['mat

print("get group min feature")
agg = df.groupby(['matchId','groupId'])[features].agg('min')
agg_rank = agg.groupby('matchId')[features].rank(pct=True).reset_index()
df_out = df_out.merge(agg.reset_index(), suffixes=["", ""], how='left', on=['matchId
df_out = df_out.merge(agg_rank, suffixes=["_min", "_min_rank"], how='left', on=['mat
print("get group size feature")
agg = df.groupby(['matchId','groupId']).size().reset_index(name='group_size')
df_out = df_out.merge(agg, how='left', on=['matchId', 'groupId'])

print("get match mean feature")
agg = df.groupby(['matchId'])[features].agg('mean').reset_index()
df_out = df_out.merge(agg, suffixes=["", "_match_mean"], how='left', on=['matchId'])

print("get match size feature")
agg = df.groupby(['matchId']).size().reset_index(name='match_size')
df_out = df_out.merge(agg, how='left', on=['matchId'])

df_out.drop(["matchId", "groupId"], axis=1, inplace=True)

X = df_out

feature_names = list(df_out.columns)

del df, df_out, agg, agg_rank
gc.collect()

return X, y, feature_names, test_idx
```

```
In [5]: x_train, y_train, train_columns, _ = feature_engineering(True,False)
```

```
processing train.csv
remove some columns
Adding Features
Removing Na's From DF
get target
get group mean feature
get group max feature
get group min feature
get group size feature
get match mean feature
get match size feature
```

```
In [7]: x_test, _, _ , test_idx = feature_engineering(False,True)
```

```
processing test.csv
remove some columns
Adding Features
Removing Na's From DF
get group mean feature
get group max feature
get group min feature
get group size feature
get match mean feature
get match size feature
```

```
In [8]: # Thanks and credited to https://www.kaggle.com/gemartin who created this wonderful mem
        def reduce_mem_usage(df):
            """ iterate through all the columns of a dataframe and modify the data type
                to reduce memory usage.
            """
            start_mem = df.memory_usage().sum()
            print('Memory usage of dataframe is {:.2f} MB'.format(start_mem))

            for col in df.columns:
                col_type = df[col].dtype

                if col_type != object:
                    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).m
                            df[col] = df[col].astype(np.float32)
                        else:
                            df[col] = df[col].astype(np.float64)
                else:
                    df[col] = df[col].astype('category')

            end_mem = df.memory_usage().sum()
            print('Memory usage after optimization is: {:.2f} MB'.format(end_mem))
            print('Decreased by {:.1f}%'.format(100 * (start_mem - end_mem) / start_mem))

            return df

        x_train = reduce_mem_usage(x_train)
        x_test = reduce_mem_usage(x_test)
```

```
Memory usage of dataframe is 4021060096.00 MB
Memory usage after optimization is: 948516192.00 MB
```

```
Decreased by 76.4%
Memory usage of dataframe is 3837401216.00 MB
Memory usage after optimization is: 903259258.00 MB
Decreased by 76.5%
```

```
In [12]: #excluded_features = []
         #use_cols = [col for col in df_train.columns if col not in excluded_features]

         train_index = round(int(x_train.shape[0]*0.8))
         dev_X = x_train[:train_index]
         val_X = x_train[train_index:]
         dev_y = y_train[:train_index]
         val_y = y_train[train_index:]
         gc.collect();

         # custom function to run light gbm model
         def run_lgb(train_X, train_y, val_X, val_y, x_test):
             params = {"objective" : "regression", "metric" : "mae", 'n_estimators':20000, 'earl
                       "num_leaves" : 31, "learning_rate" : 0.05, "bagging_fraction" : 0.7,
                        "bagging_seed" : 0, "num_threads" : 4,"colsample_bytree" : 0.7
                      }

             lgtrain = lgb.Dataset(train_X, label=train_y)
             lgval = lgb.Dataset(val_X, label=val_y)
             model = lgb.train(params, lgtrain, valid_sets=[lgtrain, lgval], early_stopping_roun

             pred_test_y = model.predict(x_test, num_iteration=model.best_iteration)
             return pred_test_y, model

         # Training the model #
         pred_test, model = run_lgb(dev_X, dev_y, val_X, val_y, x_test)
```

```
/home/sirius/anaconda3/lib/python3.6/site-packages/lightgbm/engine.py:118: UserWarning: Found `n
  warnings.warn("Found `{}` in params. Will use it instead of argument".format(alias))
/home/sirius/anaconda3/lib/python3.6/site-packages/lightgbm/engine.py:123: UserWarning: Found `e
  warnings.warn("Found `{}` in params. Will use it instead of argument".format(alias))
```

```
Training until validation scores don't improve for 200 rounds.
[1000]        training's l1: 0.0282507        valid_1's l1: 0.0288693
[2000]        training's l1: 0.0270028        valid_1's l1: 0.0281773
[3000]        training's l1: 0.026192        valid_1's l1: 0.0278832
[4000]        training's l1: 0.0255316        valid_1's l1: 0.0277071
[5000]        training's l1: 0.0249701        valid_1's l1: 0.0275958
[6000]        training's l1: 0.0244602        valid_1's l1: 0.0275161
[7000]        training's l1: 0.0240017        valid_1's l1: 0.0274589
[8000]        training's l1: 0.0235657        valid_1's l1: 0.0274077
```

```
[9000]        training's l1: 0.0231617        valid_1's l1: 0.0273648
[10000]        training's l1: 0.0227831        valid_1's l1: 0.0273305
[11000]        training's l1: 0.022418        valid_1's l1: 0.0273009
[12000]        training's l1: 0.0220675        valid_1's l1: 0.0272781
[13000]        training's l1: 0.0217335        valid_1's l1: 0.0272615
[14000]        training's l1: 0.021404        valid_1's l1: 0.0272413
[15000]        training's l1: 0.0210929        valid_1's l1: 0.0272243
[16000]        training's l1: 0.0207937        valid_1's l1: 0.0272085
[17000]        training's l1: 0.0205064        valid_1's l1: 0.0271988
[18000]        training's l1: 0.0202239        valid_1's l1: 0.0271857
[19000]        training's l1: 0.019947        valid_1's l1: 0.0271731
[20000]        training's l1: 0.0196855        valid_1's l1: 0.0271666
Did not meet early stopping. Best iteration is:
[20000]        training's l1: 0.0196855        valid_1's l1: 0.0271666
```