tryout_2

September 12, 2019

[1]:
```
# NOTE: please, look at sns.distplot for each of train/test set!
# It is really compelling to see humans vs bots distributions
```

[2]:
```
import gc
gc.collect()
```

[2]:
```
82
```

[32]:
```
# Load libs

%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns

import pandas as pd

import numpy as np

from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import classification_report
```

[4]:
```
# Load datasets

bidders = pd.read_csv('data/train.csv', header=0)
bids = pd.read_csv('data/bids.csv', header=0)
```

[39]:
```
# Training and Test sets to work with later / create features / encode etc

train = bidders.copy()
test = pd.read_csv('data/test.csv', header=0)
```

[6]:
```
# Identify one of the three periods shown on the graph above
# Use np.log for better readability

time_binning_cut = pd.cut( bids['time'], 3 )
bids['time_bin'] = time_binning_cut
```

```
bids['time_bin'] = bids['time_bin'].apply(
    lambda x: '{0:.4f}_{1:.4f}'.format( np.log(x.left), np.log(x.right) )
)

# Add feature to train and test: 'time_bin'
bids_grouped_by_bidders = bids.groupby(by='bidder_id')

bidder_no_auctions_time_bin_replacement = bids['time_bin'].value_counts().
 ↪index[1]

def apply_time_bin( bidder_id ):
    try:
        bidder_group = bids_grouped_by_bidders.get_group(bidder_id)
        time_bin_value = bidder_group['time_bin'].values[0]
        return time_bin_value
    except:
        return bidder_no_auctions_time_bin_replacement

print('train: creating time_bin...')
train['time_bin'] = train['bidder_id'].apply(
    lambda x: apply_time_bin( x )
)
print('test: creating time_bin')
test['time_bin'] = test['bidder_id'].apply(
    lambda x: apply_time_bin( x )
)
```

```
train: creating time_bin...
test: creating time_bin
```

[7]:
```
# First and last auction for each bidder
# For bidders which didn't took part in any auction: replace with random time.

bidder_no_auctions_min_time_replacement = bids['time'].median()
bidder_no_auctions_max_time_replacement =␣
 ↪bidder_no_auctions_min_time_replacement


def apply_min_time( bidder_id ):
    try:
        bidder_group = bids_grouped_by_bidders.get_group( bidder_id )
        min_time_value = bidder_group['time'].min()
        return min_time_value
    except KeyError:
        return bidder_no_auctions_min_time_replacement
```

```
def apply_max_time( bidder_id ):
    try:
        bidder_group = bids_grouped_by_bidders.get_group( bidder_id )
        max_time_value = bidder_group['time'].max()
        return max_time_value
    except KeyError:
        return bidder_no_auctions_max_time_replacement

print('train: creating min_time...')
train['min_time'] = train['bidder_id'].apply(
    lambda x: apply_min_time( x )
)
print('test: creating min_time...')
test['min_time'] = test['bidder_id'].apply(
    lambda x: apply_min_time( x )
)
print('train: creating max_time...')
train['max_time'] = train['bidder_id'].apply(
    lambda x: apply_max_time( x )
)
print('test: creating max_time...')
test['max_time'] = test['bidder_id'].apply(
    lambda x: apply_max_time( x )
)
```

```
train: creating min_time...
test: creating min_time...
train: creating max_time...
test: creating max_time...
```

[8]:
```
# Amount of bids done by single bidder

single_bidder_auctions = bids[ ['bidder_id', 'auction'] ].
 ↪groupby(by='bidder_id')
single_bidder_auctions_cnt = single_bidder_auctions.count()['auction']

print('train: creating c_bids...')
train['c_bids'] = train['bidder_id'].apply(
    lambda x: single_bidder_auctions_cnt.get(x, 0)  # 0 if no auctions found
)
print('test: creating c_bids...')
test['c_bids'] = test['bidder_id'].apply(
    lambda x: single_bidder_auctions_cnt.get(x, 0)
)
```

```
train: creating c_bids...
test: creating c_bids...
```

[9]:
```
# For each bid, calculate time passed from last bid.

bids_grouped_by_auction = bids.groupby( by='auction' )
bidid_bidresponse_mapping = {}

display('dummy: creating mapping bid_id to prev_time')
for name, group in bids_grouped_by_auction:
    group_sorted = group.sort_values(by='time')
    group_sorted['prev_time'] = group_sorted['time'].shift(1)
    time_col_idx = group_sorted.columns.get_loc('time')
    next_data_col_idx = group_sorted.columns.get_loc('prev_time')
    group_sorted.iloc[0, next_data_col_idx] = group_sorted.iloc[0, time_col_idx]
    group_sorted['bid_response'] = group_sorted['time'] -␣
 ↪group_sorted['prev_time']
    for i in group_sorted.index.values:
        bidid_bidresponse_mapping[i] = group_sorted.at[i, 'bid_response']
```

```
'dummy: creating mapping bid_id to prev_time'
```

[10]:
```
bids['bid_response'] = bids['bid_id'].apply(
    lambda bidid: np.log1p( bidid_bidresponse_mapping[bidid] )
)

# del bidid_bidresponse_mapping
```

[11]:
```
# Apply mean, min and max response for each bidder

def apply_aggfunc_bid_response( bidder_id, aggfunc, except_return=0 ):
    try:
        selected_bids = bids_grouped_by_bidders.get_group( bidder_id )
        aggfunc_log_bin_response = aggfunc( selected_bids['bid_response'] )
        return aggfunc_log_bin_response
    except KeyError:
        return except_return


print('train: mean bid_response')
train['mean_log_bid_response'] = train['bidder_id'].apply(
    lambda x: apply_aggfunc_bid_response( x, pd.Series.mean )
)
print('test: mean bid_response')
test['mean_log_bid_response'] = test['bidder_id'].apply(
    lambda x: apply_aggfunc_bid_response( x, pd.Series.mean )
)

print('train: min bid_response')
train['min_log_bid_response'] = train['bidder_id'].apply(
```

```
    lambda x: apply_aggfunc_bid_response( x, pd.Series.min )
)
print('test: min bid_response')
test['min_log_bid_response'] = test['bidder_id'].apply(
    lambda x: apply_aggfunc_bid_response( x, pd.Series.min )
)

print('train: max bid_response')
train['max_log_bid_response'] = train['bidder_id'].apply(
    lambda x: apply_aggfunc_bid_response( x, pd.Series.max )
)
print('test: max bid_response')
test['max_log_bid_response'] = test['bidder_id'].apply(
    lambda x: apply_aggfunc_bid_response( x, pd.Series.max )
)
```

```
train: mean bid_response
test: mean bid_response
train: min bid_response
test: min bid_response
train: max bid_response
test: max bid_response
```

[12]:
```
# Mark bids that were 'winning': last ones

bidid_iswinningbid_mapping = {}

print('dummy: creating bidid_iswinningbid mapping')
for name, group in bids_grouped_by_auction:
    group_sorted = group.sort_values(by='time')
    winning_bid = group_sorted.iloc[-1, :]
    bidid_iswinningbid_mapping[ winning_bid['bid_id'] ] = 1
```

```
dummy: creating bidid_iswinningbid mapping
```

[13]:
```
# Apply "bid_id -> flag_winnin_bid" mapping to bids

bids['flag_is_winning_bid'] = bids['bid_id'].apply(
    lambda x: bidid_iswinningbid_mapping.get( x, 0 )
)

# del bidid_iswinningbid_mapping
```

[14]:
```
# Total amount of wins in auctions

def apply_cwonauct( bidder_id ):
    try:
```

```
        selected_group = bids_grouped_by_bidders.get_group( bidder_id )
        return selected_group['flag_is_winning_bid'].sum()
    except KeyError:
        return 0


print('train: c_wonauct...')
train['c_wonauct'] = train['bidder_id'].apply(
    lambda x: apply_cwonauct( x )
)
print('test: c_wonauct...')
test['c_wonauct'] = test['bidder_id'].apply(
    lambda x: apply_cwonauct( x )
)
```

```
train: c_wonauct...
test: c_wonauct
```

[15]:
```
# Amount of unique countries, IPs, URLs for each bidder

def apply_distinct_colname_count( bidder_id, col_name ):
    try:
        selected_group = bids_grouped_by_bidders.get_group( bidder_id )
        distinct_colname_count = len( selected_group[col_name].unique() )
        return distinct_colname_count
    except:
        return 0


print('train: distinct countries...')
train['c_unq_countries'] = train['bidder_id'].apply(
    lambda x: apply_distinct_colname_count(x, 'country')
)
print('test: distinct countries...')
test['c_unq_countries'] = test['bidder_id'].apply(
    lambda x: apply_distinct_colname_count(x, 'country')
)

print('train: distinct IPs...')
train['c_unq_ips'] = train['bidder_id'].apply(
    lambda x: apply_distinct_colname_count(x, 'ip')
)
print('test: distinct IPs...')
test['c_unq_ips'] = test['bidder_id'].apply(
    lambda x: apply_distinct_colname_count(x, 'ip')
)
```

```
print('train: distinct URLs...')
train['c_unq_urls'] = train['bidder_id'].apply(
    lambda x: apply_distinct_colname_count(x, 'url')
)
print('test: distinct URLs...')
test['c_unq_urls'] = test['bidder_id'].apply(
    lambda x: apply_distinct_colname_count(x, 'url')
)

print('train: distinct devices...')
train['c_unq_devices'] = train['bidder_id'].apply(
    lambda x: apply_distinct_colname_count(x, 'device')
)
print('test: distinct devices...')
test['c_unq_devices'] = test['bidder_id'].apply(
    lambda x: apply_distinct_colname_count(x, 'device')
)
```

```
train: distinct countries...
test: distinct countries...
train: distinct IPs...
test: distinct IPs...
train: distinct URLs...
test: distinct URLs...
train: distinct devices...
test: distinct devices...
```

[16]:
```
# Drop redundant features

train = train.drop(['payment_account', 'address'], axis=1)
test = test.drop(['payment_account', 'address'], axis=1)

# display(train.head())
# display(test.head())
```

[17]:
```
# Prepare the X and y to fit classifier

train_ids = train['bidder_id']
test_ids = test['bidder_id']

train_labels = train['outcome']

train = train.drop(['bidder_id', 'outcome'], axis=1)
test = test.drop(['bidder_id'], axis=1)

# Encode time_bin column
timebin_lbl_encoder = LabelEncoder()
train['time_bin'] = timebin_lbl_encoder.fit_transform( train['time_bin'] )
```

```
test['time_bin'] = timebin_lbl_encoder.transform( test['time_bin'] )

# Log-transform max_time and min_time columns: other time-related columns are␣
 ↪log-transformed too
train['min_time'] = np.log1p( train['min_time'] )
train['max_time'] = np.log1p( train['max_time'] )
test['min_time'] = np.log1p( test['min_time'] )
test['max_time'] = np.log1p( test['max_time'] )
```

[18]:
```
train.head()
```

[18]:
```
   time_bin   min_time   max_time  c_bids  mean_log_bid_response  \
0         2  36.817016  36.818361      24              22.872016
1         2  36.817483  36.818145       3              18.376080
2         2  36.817622  36.818353       4              23.044771
3         1  36.811566  36.811566       1              23.577920
4         2  36.816992  36.818218     155              18.222390

   min_log_bid_response  max_log_bid_response  c_wonauct  c_unq_countries  \
0              0.000000             29.028216          0                6
1             17.778827             19.570586          0                1
2             22.148275             23.583962          0                1
3             23.577920             23.577920          0                1
4              0.000000             25.519491          0                2

   c_unq_ips  c_unq_urls  c_unq_devices
0         20           1             14
1          3           2              2
2          4           2              2
3          1           1              1
4        123          91             53
```

[19]:
```
test.head()
```

[19]:
```
   time_bin   min_time   max_time  c_bids  mean_log_bid_response  \
0         0  36.803984  36.811248       4              14.712477
1         0  36.803910  36.811769       3              22.140747
2         1  36.811428  36.811458      17              20.826804
3         0  36.803863  36.811776     148              21.415145
4         0  36.804271  36.804953      23              24.570265

   min_log_bid_response  max_log_bid_response  c_wonauct  c_unq_countries  \
0              0.000000             21.905961          0                3
1             18.471974             24.484466          0                2
2              0.000000             24.447055          0                3
3              0.000000             31.696460          0               14
4             20.551416             29.216565          0                2

   c_unq_ips  c_unq_urls  c_unq_devices
```

```
0          4          3              2
1          2          1              3
2          4          2              4
3        129         80             81
4         17          1             17
```

[27]:
```
# Build the model

X_train = train
y_train = train_labels

rfc_model = RandomForestClassifier(
    n_estimators=1000, max_depth=25, min_samples_leaf=3,
    n_jobs=-1, verbose=1
)

X_tr, X_val, y_tr, y_val = train_test_split( X_train, y_train, test_size=0.35,␣
 ↪shuffle=True )

rfc_model.fit( X_tr, y_tr )
```

```
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:    0.0s
[Parallel(n_jobs=-1)]: Done 184 tasks      | elapsed:    0.1s
[Parallel(n_jobs=-1)]: Done 434 tasks      | elapsed:    0.3s
[Parallel(n_jobs=-1)]: Done 784 tasks      | elapsed:    0.6s
[Parallel(n_jobs=-1)]: Done 1000 out of 1000 | elapsed:    0.7s finished
```

[27]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=25, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=3, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=1000,
                       n_jobs=-1, oob_score=False, random_state=None, verbose=1,
                       warm_start=False)
```

[34]:
```
# report

y_pred = rfc_model.predict( X_val )

print( classification_report( y_val, y_pred ) )

# .score
display( rfc_model.score(X_val, y_val) )
```

```
[Parallel(n_jobs=8)]: Using backend ThreadingBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done 184 tasks      | elapsed:    0.1s
```

```
[Parallel(n_jobs=8)]: Done 434 tasks      | elapsed:    0.1s
[Parallel(n_jobs=8)]: Done 784 tasks      | elapsed:    0.2s
[Parallel(n_jobs=8)]: Done 1000 out of 1000 | elapsed:    0.2s finished

              precision    recall  f1-score   support

         0.0       0.95      1.00      0.97       668
         1.0       0.50      0.05      0.10        37

    accuracy                           0.95       705
   macro avg       0.73      0.53      0.54       705
weighted avg       0.93      0.95      0.93       705


[Parallel(n_jobs=8)]: Using backend ThreadingBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done 184 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done 434 tasks      | elapsed:    0.1s
[Parallel(n_jobs=8)]: Done 784 tasks      | elapsed:    0.1s
[Parallel(n_jobs=8)]: Done 1000 out of 1000 | elapsed:    0.2s finished

0.9475177304964539
```

[24]:
```
# CV
cv_scores = cross_val_score(rfc_model, X_train, y_train, cv=10)

# display(rfc_model.score())
display(cv_scores.mean())
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    0.1s finished
[Parallel(n_jobs=8)]: Using backend ThreadingBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done 100 out of 100 | elapsed:    0.0s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    0.1s finished
[Parallel(n_jobs=8)]: Using backend ThreadingBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done 100 out of 100 | elapsed:    0.0s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    0.1s finished
[Parallel(n_jobs=8)]: Using backend ThreadingBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done 100 out of 100 | elapsed:    0.0s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    0.2s finished
[Parallel(n_jobs=8)]: Using backend ThreadingBackend with 8 concurrent workers.
```

```
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done 100 out of 100 | elapsed:    0.0s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    0.2s finished
[Parallel(n_jobs=8)]: Using backend ThreadingBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done 100 out of 100 | elapsed:    0.0s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    0.1s finished
[Parallel(n_jobs=8)]: Using backend ThreadingBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done 100 out of 100 | elapsed:    0.0s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    0.1s finished
[Parallel(n_jobs=8)]: Using backend ThreadingBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done 100 out of 100 | elapsed:    0.0s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    0.1s finished
[Parallel(n_jobs=8)]: Using backend ThreadingBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done 100 out of 100 | elapsed:    0.0s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    0.2s finished
[Parallel(n_jobs=8)]: Using backend ThreadingBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done 100 out of 100 | elapsed:    0.0s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:    0.1s finished
[Parallel(n_jobs=8)]: Using backend ThreadingBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done 100 out of 100 | elapsed:    0.0s finished
```

```
0.9518077927195703
```

[30]:
```
display(
    pd.DataFrame(
        {'col_name': X_train.columns.values, 'feat_imp': rfc_model.
 ↪feature_importances_}
    ).sort_values(by='feat_imp')
)
```

	col_name	feat_imp
5	min_log_bid_response	0.007526
0	time_bin	0.009986
7	c_wonauct	0.049518
4	mean_log_bid_response	0.067592

```
8         c_unq_countries  0.075012
1                min_time  0.078072
2                max_time  0.084903
6    max_log_bid_response  0.091661
10             c_unq_urls  0.107534
9               c_unq_ips  0.113997
11          c_unq_devices  0.133893
3                   c_bids  0.180307
```

[38]:
```
# Create a submission

test_pred = rfc_model.predict_proba( test )

submission = pd.DataFrame({
    'bidder_id': test_ids,
    'prediction': test_pred[:, 1]
})

submission.to_csv('submission.csv', index=False)
```

```
[Parallel(n_jobs=8)]: Using backend ThreadingBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done  34 tasks      | elapsed:    0.0s
[Parallel(n_jobs=8)]: Done 184 tasks      | elapsed:    0.1s
[Parallel(n_jobs=8)]: Done 434 tasks      | elapsed:    0.1s
[Parallel(n_jobs=8)]: Done 784 tasks      | elapsed:    0.2s
[Parallel(n_jobs=8)]: Done 1000 out of 1000 | elapsed:    0.3s finished
```