Ran SONG - Feature Engineering and Model Training

January 19, 2022

#

Feature Engineering and Model Training

0.1 1. Feature Engineering

0.1.1 1. General Feature Creation

1.1 How to create a generic composite feature The general combined feature refers to the feature created by counting the sum of different discrete features at different value levels and different continuous feature values, and grouping and summing them according to `card_id`. The specific creation process can be understood by the following simple example:

The data set created by this method can not only represent the consumption of each `card_id` from as many dimensions as possible, but also can be smoothly spliced with the training set/test set, so that it can be brought into the model for modeling. We can use the dictionary object type in Python to implement the related process. The implementation process of the above simple example is as follows:

[2]:
```
import gc
import time
import numpy as np
import pandas as pd
from datetime import datetime
```

[5]:
```
t1 = {'card_id':[1, 2, 1, 3],
      'A':[1, 2, 1, 2],
      'B':[2, 1, 2, 2],
      'C':[4, 5, 1, 5],
      'D':[7, 5, 4, 8],}

t1 = pd.DataFrame(t1)
t1
```

[5]:

	card_id	A	B	C	D
0	1	1	2	4	7
1	2	2	1	5	5
2	1	1	2	1	4
3	3	2	2	5	8

[15]:
```
numeric_cols = ['C', 'D']
category_cols = ['A', 'B']
```

[9]:
```
features = {}
card_all = t1['card_id'].values.tolist()
for card in card_all:
    features[card] = {}
```

[10]:
```
features
```

[10]:
```
{1: {}, 2: {}, 3: {}}
```

[11]:
```
columns = t1.columns.tolist()
columns
```

[11]:
```
['card_id', 'A', 'B', 'C', 'D']
```

[12]:
```
idx = columns.index('card_id')
idx
```

[12]:
```
0
```

[16]:
```
category_cols_index = [columns.index(col) for col in category_cols]
category_cols_index
```

[16]:
```
[1, 2]
```

[17]:
```
numeric_cols_index = [columns.index(col) for col in numeric_cols]
numeric_cols_index
```

[17]:
```
[3, 4]
```

[19]:
```
# Combination of different values of discrete fields and continuous fields
# Complete group summation at the same time
for i in range(t1.shape[0]):
    va = t1.loc[i].values
    card = va[idx]
    for cate_ind in category_cols_index:
        for num_ind in numeric_cols_index:
            col_name = '&'.join([columns[cate_ind], str(va[cate_ind]),␣
 ↪columns[num_ind]])
            features[card][col_name] = features[card].get(col_name, 0) +␣
 ↪va[num_ind]
```

[26]:
```
features
```

[26]:
```
{1: {'A&1&C': 5, 'A&1&D': 11, 'B&2&C': 5, 'B&2&D': 11},
 2: {'A&2&C': 5, 'A&2&D': 5, 'B&1&C': 5, 'B&1&D': 5},
 3: {'A&2&C': 5, 'A&2&D': 8, 'B&2&C': 5, 'B&2&D': 8}}
```

It can be found that at this time features is a grouped summation result under different `card_id` after combining new features with different values of discrete variables and continuous variables. Next we convert it to a DataFrame:

[20]:
```
df = pd.DataFrame(features).T.reset_index()

cols = df.columns.tolist()

df.columns = ['card_id'] + cols[1:]
df
```

[20]:

	card_id	A&1&C	A&1&D	B&2&C	B&2&D	A&2&C	A&2&D	B&1&C	B&1&D
0	1	10.0	22.0	10.0	22.0	NaN	NaN	NaN	NaN
1	2	NaN	NaN	NaN	NaN	10.0	10.0	10.0	10.0
2	3	NaN	NaN	10.0	16.0	10.0	16.0	NaN	NaN

It is not difficult to find through the above process that this method of feature creation can very efficiently represent hidden information in more data sets, but this method is prone to generate many null values, and it is necessary to consider that the feature matrix is too sparse in the subsequent modeling process resulting problems.

1.2 Create generic composite features based on transaction datasets

It should be noted here that due to the large transaction dataset itself, although the summation part of the feature creation work will reduce the volume of data that is finally brought into the modeling to a certain extent, the operation of the transaction dataset itself requires a lot of content and content. For a certain period of time, if you want to manually execute the following code, it is recommended to configure at least 32G memory and above.

- Read in data

`transaction_d_pre.csv`

[65]:
```
train = pd.read_csv('preprocess/train_pre.csv')
test =  pd.read_csv('preprocess/test_pre.csv')
transaction = pd.read_csv('preprocess/transaction_d_pre.csv')
```

[21]:
```
# discrete vs contuniuous
numeric_cols = ['purchase_amount', 'installments']

category_cols = ['authorized_flag', 'city_id', 'category_1',
       'category_3',␣
 ↪'merchant_category_id','month_lag','most_recent_sales_range',
                'most_recent_purchases_range', 'category_4',
```

```
                'purchase_month', 'purchase_hour_section', 'purchase_day']

id_cols = ['card_id', 'merchant_id']
```

- feature engineering

[7]:
```
features = {}
card_all = train['card_id'].append(test['card_id']).values.tolist()
for card in card_all:
    features[card] = {}

# tag indexes
columns = transaction.columns.tolist()
idx = columns.index('card_id')
category_cols_index = [columns.index(col) for col in category_cols]
numeric_cols_index = [columns.index(col) for col in numeric_cols]

# processing time
s = time.time()
num = 0

# record time
for i in range(transaction.shape[0]):
    va = transaction.loc[i].values
    card = va[idx]
    for cate_ind in category_cols_index:
        for num_ind in numeric_cols_index:
            col_name = '&'.join([columns[cate_ind], va[cate_ind],␣
 ↪columns[num_ind]])
            features[card][col_name] = features[card].get(col_name, 0) +␣
 ↪va[num_ind]
    num += 1
    if num%1000000==0:
        print(time.time()-s, "s")
del transaction
gc.collect()
```

```
142.746732711792 s
241.50783610343933 s
338.9149408340454 s
436.4667372703552 s
533.113107919693 s
629.66761469841 s
727.1969571113586 s
824.3946213722229 s
921.0717754364014 s
1017.7034878730774 s
```

```
1114.4361855983734 s
1211.2046930789948 s
1308.0264575481415 s
1404.8067374229431 s
1501.533932209015 s
1598.396145105362 s
1695.2529389858246 s
1792.5994687080383 s
1889.7299542427063 s
1987.0093190670013 s
2084.849946975708 s
2183.5546836853027 s
2281.704159259796 s
2379.819750070572 s
2478.2387039661407 s
2576.8626248836517 s
2676.383053302765 s
2777.3995122909546 s
2879.5466351509094 s
2981.8099772930145 s
3085.8226635456085 s
```

[7]:
```
0
```

Inout training & validation set

[8]:
```
df = pd.DataFrame(features).T.reset_index()
del features
cols = df.columns.tolist()
df.columns = ['card_id'] + cols[1:]

# generate datasets
train = pd.merge(train, df, how='left', on='card_id')
test =  pd.merge(test, df, how='left', on='card_id')
del df
train.to_csv("preprocess/train_dict.csv", index=False)
test.to_csv("preprocess/test_dict.csv", index=False)

gc.collect()
```

0.1.2 2.

In addition to general combined features, we can also consider feature extraction from another perspective, that is, first grouping according to card_id, then counting different fields and related statistics in each group, and then using it as Features, brought in for modeling. Its basic structural features are as follows:

The process is not complicated and can be implemented quickly through the groupby process in pandas. Different from the previous idea of feature construction, the features constructed by this

method will not have a large number of missing values, and there will be relatively few new columns. The code implementation process is as follows:

- Read in data

[36]:
```
transaction = pd.read_csv('preprocess/transaction_g_pre.csv')
```

- Tag ids

[22]:
```
# contunious vs discrete
numeric_cols = ['authorized_flag',  'category_1', 'installments',
       'category_3', ␣
 ↪'month_lag','purchase_month','purchase_day','purchase_day_diff',␣
 ↪'purchase_month_diff',
       'purchase_amount', 'category_2',
       'purchase_month', 'purchase_hour_section', 'purchase_day',
       'most_recent_sales_range', 'most_recent_purchases_range', 'category_4']
categorical_cols = ['city_id', 'merchant_category_id', 'merchant_id',␣
 ↪'state_id', 'subsector_id']
```

- feature engineering

[7]:
```
aggs = {}

for col in numeric_cols:
    aggs[col] = ['nunique', 'mean', 'min', 'max','var','skew', 'sum']
for col in categorical_cols:
    aggs[col] = ['nunique']
aggs['card_id'] = ['size', 'count']
cols = ['card_id']

for key in aggs.keys():
    cols.extend([key+'_'+stat for stat in aggs[key]])

df = transaction[transaction['month_lag']<0].groupby('card_id').agg(aggs).
 ↪reset_index()
df.columns = cols[:1] + [co+'_hist' for co in cols[1:]]

df2 = transaction[transaction['month_lag']>=0].groupby('card_id').agg(aggs).
 ↪reset_index()
df2.columns = cols[:1] + [co+'_new' for co in cols[1:]]
df = pd.merge(df, df2, how='left',on='card_id')

df2 = transaction.groupby('card_id').agg(aggs).reset_index()
df2.columns = cols
df = pd.merge(df, df2, how='left',on='card_id')
del transaction
gc.collect()
```

```
train = pd.merge(train, df, how='left', on='card_id')
test =  pd.merge(test, df, how='left', on='card_id')
del df
train.to_csv("preprocess/train_groupby.csv", index=False)
test.to_csv("preprocess/test_groupby.csv", index=False)

gc.collect()
```

Information after data generation

0.1.3 3. Merging data

The aforementioned features are still stored in different data files. We need to merge them before they can be further brought in for modeling. The merging process is relatively simple, just need to combine `train_dict(test_dict)` with `train_group(test_group)` Perform horizontal splicing according to `card_id`, and then remove duplicate columns

[2]:
```
train_dict = pd.read_csv("preprocess/train_dict.csv")
test_dict = pd.read_csv("preprocess/test_dict.csv")
train_groupby = pd.read_csv("preprocess/train_groupby.csv")
test_groupby = pd.read_csv("preprocess/test_groupby.csv")
```

- Remove duplicate columns

[3]:
```
for co in train_dict.columns:
    if co in train_groupby.columns and co!='card_id':
        del train_groupby[co]
for co in test_dict.columns:
    if co in test_groupby.columns and co!='card_id':
        del test_groupby[co]
```

[4]:
```
train = pd.merge(train_dict, train_groupby, how='left', on='card_id').fillna(0)
test = pd.merge(test_dict, test_groupby, how='left', on='card_id').fillna(0)
```

> Note that the above operation fills the missing values with 0. The missing values here are not real missing values. The missing values are just values that have no statistical results during the feature creation process. Logically, these values are actually 0. Therefore, missing value filling here is equivalent to data completion.

- Memory Management

[5]:
```
train.to_csv("preprocess/train.csv", index=False)
test.to_csv("preprocess/test.csv", index=False)

del train_dict, test_dict, train_groupby, test_groupby
gc.collect()
```

[5]: 352

0.2 2. Random Forest Prediction

[5]:
```
train = pd.read_csv("preprocess/train.csv")
test = pd.read_csv("preprocess/test.csv")
```

- feature selection

Since thousands of features have been created before, if all the features are used for modeling, the modeling time of the model will be greatly prolonged, and too many irrelevant features are brought in to improve the model results, so here we use the Pearson correlation coefficient, select the 300 features most relevant to the label for modeling. Of course, the 300 here can also be adjusted by itself.

[ ]:
```
features = train.columns.tolist()
features.remove("card_id")
features.remove("target")
featureSelect = features[:]

# correlation stats
corr = []
for fea in featureSelect:
    corr.append(abs(train[[fea, 'target']].fillna(0).corr().values[0][1]))

# create model by top 300
se = pd.Series(corr, index=featureSelect).sort_values(ascending=False)
feature_select = ['card_id'] + se[:300].index.tolist()

# output result
train = train[feature_select + ['target']]
test = test[feature_select]
```

Note that the main reason why feature extraction can be performed by the Pearson correlation coefficient is that we default all features to continuous variables in the process of feature creation

- Parameter tuning with grid search

`Gridsearch` from `sklearn`

Import the relevant packages, including the mean square error calculation function, the random forest estimator and the grid search estimator:

[30]:
```
from sklearn.metrics import mean_squared_error
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV
```

Name	Description
criterion	Rule evaluation index or loss function, default Gini coefficient, optional information entropy

Name	Description
splitter	The tree model growth method, the default is to grow the fastest way to reduce the value of the loss function, and it can be randomly divided according to a certain condition.
max_depth	map_depth for iteration in trees
min_samples_split	Minimum number of samples required for internal node subdivision
min_samples_leaf	Leaf nodes contain the minimum number of samples
min_weight_fraction_leaf	Minimum weights required for leaf nodes and
max_features	How many features can be brought in at most for segmentation rule selection during segmentation
random_state	random seed
max_leaf_nodes	max leaf node number
min_impurity_decrease	Data set subdivision needs to reduce the loss value at least
min_impurity_split	Minimum impurity required for dataset subdivision, will be removed in version 0.25
class_weight	Various sample weights

search by `n_estimators`,`min_samples_leaf`,`min_samples_split`,`max_depth`,`max_features`:

It is recommended to use `RandomizedSearchCV` to determine the approximate range, and then use `GridSearchCV` to search for specific parameter values with high precision

[43]:
```
features = train.columns.tolist()
features.remove("card_id")
features.remove("target")


parameter_space = {
    "n_estimators": [79, 80, 81],
    "min_samples_leaf": [29, 30, 31],
    "min_samples_split": [2, 3],
    "max_depth": [9, 10],
    "max_features": ["auto", 80]
}
```

[44]:
```
clf = RandomForestRegressor(
    criterion="mse",
    n_jobs=15,
    random_state=22)
```

[45]:
```
grid = GridSearchCV(clf, parameter_space, cv=2,␣
 ↪scoring="neg_mean_squared_error")
grid.fit(train[features].values, train['target'].values)
```

[45]:
```
GridSearchCV(cv=2, estimator=RandomForestRegressor(n_jobs=15, random_state=22),
             param_grid={'max_depth': [9, 10], 'max_features': ['auto', 80],
                         'min_samples_leaf': [29, 30, 31],
                         'min_samples_split': [2, 3],
                         'n_estimators': [79, 80, 81]},
             scoring='neg_mean_squared_error')
```

Check results:

[46]:
```
grid.best_params_
```

[46]:
```
{'max_depth': 10,
 'max_features': 80,
 'min_samples_leaf': 31,
 'min_samples_split': 2,
 'n_estimators': 80}
```

At the same time, we can also directly view or call the optimal parameter composition estimator:

[47]:
```
grid.best_estimator_
```

[47]:
```
RandomForestRegressor(max_depth=10, max_features=80, min_samples_leaf=31,
                      n_estimators=80, n_jobs=15, random_state=22)
```

Final score on training set:

[48]:
```
np.sqrt(-grid.best_score_)
```

[48]:
```
3.6900889856014247
```

[49]:
```
grid.best_estimator_.predict(test[features])
```

[49]:
```
array([-3.42895506, -1.05271922, -0.34647055, …,  0.71331227,
       -2.40402906,  0.29249733])
```

Export to CSV

[50]:
```
test['target'] = grid.best_estimator_.predict(test[features])
test[['card_id', 'target']].to_csv("result/submission_randomforest.csv",␣
 ↪index=False)
```