import numpy as np

import pandas as pd

from time import time

import pprint

import joblib

from functools import partial

*# Suppressing warnings because of skopt verbosity*

import warnings

warnings.filterwarnings("ignore")

*# Classifier/Regressor*

from xgboost import XGBRegressor, DMatrix

*# Model selection*

from sklearn.model\_selection import KFold, StratifiedKFold

*# Metrics*

from sklearn.metrics import mean\_squared\_error

from sklearn.metrics import make\_scorer

*# Skopt functions*

from skopt import BayesSearchCV

from skopt.callbacks import DeadlineStopper, DeltaYStopper

from skopt.space import Real, Categorical, Integer

*# Data processing*

from sklearn.preprocessing import OrdinalEncoder

from sklearn.decomposition import TruncatedSVD

from sklearn.feature\_extraction.text import TfidfVectorizer, CountVectorizer

In [4]:

*# Loading data*

X\_train = pd.read\_csv("../input/30-days-of-ml/train.csv")

X\_test = pd.read\_csv("../input/30-days-of-ml/test.csv")

In [5]:

*# Preparing data as a tabular matrix*

y\_train = X\_train.target

X\_train = X\_train.set\_index('id').drop('target', axis='columns')

X\_test = X\_test.set\_index('id')

In [6]:

*# Stratifying the target*

y\_stratified = pd.cut(y\_train.rank(method='first'), bins=10, labels=False)

In [7]:

*# Winsorizing lower bounds*

from scipy.stats.mstats import winsorize

y\_train = np.array(winsorize(y\_train, [0.002, 0.0]))

In [8]:

*# Pointing out categorical features*

categoricals = [item for item **in** X\_train.columns if 'cat' **in** item]

In [9]:

*# Dealing with categorical data using get\_dummies*

dummies = pd.get\_dummies(X\_train.append(X\_test)[categoricals])

X\_train[dummies.columns] = dummies.iloc[:len(X\_train), :]

X\_test[dummies.columns] = dummies.iloc[len(X\_train): , :]

del(dummies)

In [10]:

*# Dealing with categorical data using OrdinalEncoder*

ordinal\_encoder = OrdinalEncoder()

X\_train[categoricals] = ordinal\_encoder.fit\_transform(X\_train[categoricals])

X\_test[categoricals] = ordinal\_encoder.transform(X\_test[categoricals])

In [11]:

*# Feature selection (https://www.kaggle.com/lucamassaron/tutorial-feature-selection-with-boruta-shap)*

important\_features = ['cat8\_E', 'cont0', 'cont5', 'cont7', 'cont8', 'cat1\_A', 'cont2', 'cont13',

'cont3', 'cont10', 'cont1', 'cont9', 'cont11', 'cat1', 'cat8\_C', 'cont6',

'cont12', 'cat5', 'cat3\_C', 'cont4', 'cat8']

X\_train = X\_train[important\_features]

X\_test = X\_test[important\_features]

Setting up optimization

First, we create a wrapper function to deal with running the optimizer and reporting back its best results.

In [12]:

*# Reporting util for different optimizers*

def report\_perf(optimizer, X, y, title="model", callbacks=None):

*"""*

*A wrapper for measuring time and performances of different optmizers*

*optimizer = a sklearn or a skopt optimizer*

*X = the training set*

*y = our target*

*title = a string label for the experiment*

*"""*

start = time()

if callbacks **is** **not** None:

optimizer.fit(X, y, callback=callbacks)

else:

optimizer.fit(X, y)

d=pd.DataFrame(optimizer.cv\_results\_)

best\_score = optimizer.best\_score\_

best\_score\_std = d.iloc[optimizer.best\_index\_].std\_test\_score

best\_params = optimizer.best\_params\_

print((title + " took **%.2f** seconds, candidates checked: **%d**, best CV score: **%.3f** "

+ u"**\u00B1**"+" **%.3f**") % (time() - start,

len(optimizer.cv\_results\_['params']),

best\_score,

best\_score\_std))

print('Best parameters:')

pprint.pprint(best\_params)

print()

return best\_params

We then define the evaluation metric, using the Scikit-learn function make\_scorer allows us to convert the optimization into a minimization problem, as required by Scikit-optimize. We set squared=False by means of a partial function to obtain the root mean squared error (RMSE) as evaluation.

In [13]:

*# Setting the scoring function*

scoring = make\_scorer(partial(mean\_squared\_error, squared=False),

greater\_is\_better=False)

We set up a 7-fold cross validation

In [14]:

*# Setting the validation strategy*

skf = StratifiedKFold(n\_splits=7,

shuffle=True,

random\_state=0)

cv\_strategy = list(skf.split(X\_train, y\_stratified))

We set up a generic XGBoost regressor.

In [15]:

*# Setting the basic regressor*

reg = XGBRegressor(random\_state=0, booster='gbtree', objective='reg:squarederror', tree\_method='gpu\_hist')

We define a search space, expliciting the key hyper-parameters to optimize and the range where to look for the best values.

In [16]:

*# Setting the search space*

search\_spaces = {'learning\_rate': Real(0.01, 1.0, 'uniform'),

'max\_depth': Integer(2, 12),

'subsample': Real(0.1, 1.0, 'uniform'),

'colsample\_bytree': Real(0.1, 1.0, 'uniform'), *# subsample ratio of columns by tree*

'reg\_lambda': Real(1e-9, 100., 'uniform'), *# L2 regularization*

'reg\_alpha': Real(1e-9, 100., 'uniform'), *# L1 regularization*

'n\_estimators': Integer(50, 5000)

}

We then define the Bayesian optimization engine, providing to it our XGBoost, the search spaces, the evaluation metric, the cross-validation. We set a large number of possible experiments and some parallelism in the search operations.

In [17]:

*# Wrapping everything up into the Bayesian optimizer*

opt = BayesSearchCV(estimator=reg,

search\_spaces=search\_spaces,

scoring=scoring,

cv=cv\_strategy,

n\_iter=120, *# max number of trials*

n\_points=1, *# number of hyperparameter sets evaluated at the same time*

n\_jobs=1, *# number of jobs*

iid=False, *# if not iid it optimizes on the cv score*

return\_train\_score=False,

refit=False,

optimizer\_kwargs={'base\_estimator': 'GP'}, *# optmizer parameters: we use Gaussian Process (GP)*

random\_state=0) *# random state for replicability*

Finally we runt the optimizer and wait for the results. We have set some limits to its operations: we required it to stop if it cannot get consistent improvements from the search (DeltaYStopper) and time dealine set in seconds (we decided for 6 hours).

In [18]:

*# Running the optimizer*

overdone\_control = DeltaYStopper(delta=0.0001) *# We stop if the gain of the optimization becomes too small*

time\_limit\_control = DeadlineStopper(total\_time=60\*60\*4) *# We impose a time limit (7 hours)*

best\_params = report\_perf(opt, X\_train, y\_train,'XGBoost\_regression',

callbacks=[overdone\_control, time\_limit\_control])

XGBoost\_regression took 11287.00 seconds, candidates checked: 62, best CV score: -0.709 ± 0.000

Best parameters:

OrderedDict([('colsample\_bytree', 0.13641457979856397),

('learning\_rate', 0.09604083138419779),

('max\_depth', 5),

('n\_estimators', 2206),

('reg\_alpha', 68.85614181114505),

('reg\_lambda', 9.260067417192285),

('subsample', 0.8980218163372579)])

Prediction on test data

Having got the best hyperparameters for the data at hand, we instantiate a XGBoost using such values and train our model on all the available examples.

After having trained the model, we predict on the test set and we save the results on a csv file.

In [19]:

*# Transferring the best parameters to our basic regressor*

reg = XGBRegressor(random\_state=0, booster='gbtree', objective='reg:squarederror', tree\_method='gpu\_hist', \*\*best\_params)

In [20]:

*# Cross-validation prediction*

folds = 10

skf = StratifiedKFold(n\_splits=folds,

shuffle=True,

random\_state=0)

predictions = np.zeros(len(X\_test))

rmse = list()

for k, (train\_idx, val\_idx) **in** enumerate(skf.split(X\_train, y\_stratified)):

reg.fit(X\_train.iloc[train\_idx, :], y\_train[train\_idx])

val\_preds = reg.predict(X\_train.iloc[val\_idx, :])

val\_rmse = mean\_squared\_error(y\_true=y\_train[val\_idx], y\_pred=val\_preds, squared=False)

print(f"Fold **{**k**}** RMSE: **{**val\_rmse**:**0.5f**}**")

rmse.append(val\_rmse)

predictions += reg.predict(X\_test).ravel()

predictions /= folds

print(f"repeated CV RMSE: **{**np.mean(rmse)**:**0.5f**}** (std=**{**np.std(rmse)**:**0.5f**}**)")

Fold 0 RMSE: 0.70777

Fold 1 RMSE: 0.71033

Fold 2 RMSE: 0.70861

Fold 3 RMSE: 0.70917

Fold 4 RMSE: 0.70706

Fold 5 RMSE: 0.70931

Fold 6 RMSE: 0.71208

Fold 7 RMSE: 0.70771

Fold 8 RMSE: 0.70977

Fold 9 RMSE: 0.70757

repeated CV RMSE: 0.70894 (std=0.00145)

In [21]:

*# Preparing the submission*

submission = pd.DataFrame({'id':X\_test.index,

'target': predictions})

submission.to\_csv("submission.csv", index = False)

In [22]:

submission

Out[22]:

|  | id | target |
| --- | --- | --- |
| 0 | 0 | 8.145761 |
| 1 | 5 | 8.392107 |
| 2 | 15 | 8.405922 |
| 3 | 16 | 8.532095 |
| 4 | 17 | 8.160963 |
| ... | ... | ... |
| 199995 | 499987 | 8.027081 |
| 199996 | 499990 | 8.451505 |
| 199997 | 499991 | 8.457389 |
| 199998 | 499994 | 8.156412 |
| 199999 | 499995 | 7.992031 |

200000 rows × 2 columns

In [ ]: