import numpy as np

from sklearn.model\_selection import cross\_val\_score

from catboost import CatBoostRegressor

# set output float precision

np.set\_printoptions(precision=3)

# init model

model = CatBoostRegressor(random\_seed=0, verbose=False)

# calculate mae on folds

mae = cross\_val\_score(model, data[best\_features], data['target'],

cv=5, scoring='neg\_mean\_absolute\_error', n\_jobs=8)

# print the results

print('folds: {}'.format(abs(mae)))

print('total: {:.3f}'.format(np.mean(abs(mae))))

CatboostRegressor (without any tuning) trained on 15 features having highest importance score demonstrates mean average error 2.064.

folds: [1.982 2.333 2.379 1.266 2.362]

total: 2.064

Feature selection

To avoid a potential overfitting, we employ a genetic algorithm for feature selection. The genetic context is pretty straightforward. We suppose that the list of features (without duplicates) is the chromosome, whereas each gene represents one feature. n\_features is the input parameter controlling the amount of genes in the chromosome.

import random

class Chromosome(object):

def \_init\_(self, genes, size):

self.genes = random.sample(genes, size)

We generate the population with 50 chromosomes, where each gene is generated as a random choice from initial list of features (1496 features). To accelerate the performance, we also add to population the feature set used in the baseline model.

from deap import base, creator, tools

def init\_individual(ind\_class, genes=None, size=None):

return ind\_class(genes, size)

genes = [

column for column in train.columns

if column not in ['target', 'seg\_id']

]

# setting individual creator

creator.create('FitnessMin', base.Fitness, weights=(-1,))

creator.create('Individual', Chromosome, fitness=creator.FitnessMin)

# register callbacks

toolbox = base.Toolbox()

toolbox.register(

'individual', init\_individual, creator.Individual,

genes=genes, size=n\_features)

toolbox.register(

'population', tools.initRepeat, list, toolbox.individual)

# raise population

pop = toolbox.population(50)

Standard two-point crossover operator is used for crossing two chromosomes.

toolbox.register('mate', tools.cxTwoPoint)

To implement a mutation, we first generate a random amount of genes (> 1), which needs to be mutated, and then mutate these genes in order that the chromosome doesn't contain two equal genes.

Note, that mutation operator must return a tuple.

def mutate(individual, genes=None, pb=0):

# maximal amount of mutated genes

n\_mutated\_max = max(1, int(len(individual) \* pb))

# generate the random amount of mutated genes

n\_mutated = random.randint(1, n\_mutated\_max)

# select random genes which need to be mutated

mutated\_indexes = random.sample(

[index for index in range(len(individual.genes))], n\_mutated)

# mutation

for index in mutated\_indexes:

individual[index] = random.choice(genes)

return individual,

toolbox.register('mutate', mutate, genes=genes, pb=0.2)

For fitness evaluation we use lightened version of CatboostRegressor with decreased number of iterations and increased learning rate. Note, that fitness evaluator must also return a tuple.

from catboost import CatBoostRegressor

from sklearn.model\_selection import cross\_val\_score

model = CatBoostRegressor(

iterations=60, learning\_rate=0.2, random\_seed=0, verbose=False)

def evaluate(individual, model=None, train=None, n\_splits=5):

mae\_folds = cross\_val\_score(

model,

train[individual.genes],

train['target'],

cv=n\_splits,

scoring='neg\_mean\_absolute\_error')

return abs(mae\_folds.mean()),

from operator import attrgetter

def select\_best(individuals, k, fit\_attr='fitness'):

return sorted(set(individuals), key=attrgetter(fit\_attr), reverse=True)[:k]

toolbox.register('select', select\_best)

hof = tools.HallOfFame(5)

from deap import algorithms

algorithms.eaMuPlusLambda(

pop, toolbox,

mu=10, lambda\_=30, cxpb=0.2, mutpb=0.8,

ngen=50, stats=stats, halloffame=hof, verbose=True)