Earthquake prediction using python

AI_phase 4

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()),

toolbox.register(
    'evaluate', evaluate, model=model, train=train, n_splits=5)
```

We register elitism operator to select best individuals to the next generation. The amount of the best individuals is controlling by the parameter mu in the algorithm. To prevent populations with many duplicate individuals, we overwrite the standard selBest operator.

```
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)
```

To keep track of the best individuals, we introduce a hall of fame container.

```
hof = tools.HallOfFame(5)
```