

# Tarefa\_4

May 3, 2021

## 1 Tarefa 4

Alunos: Andreza (164213), Gil (225323) e Yan (118982)

```
[5]: import time
import numpy as np
from scipy.stats import loguniform, uniform

from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error, make_scorer
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV,
    ↪cross_val_score

from hyperopt import tpe, fmin, hp, STATUS_OK
from pyswarm import pso
from simanneal import Annealer
import optuna

from IPython.display import display, clear_output
```

### 1.1 Hiperparâmetros:

- C:  $2^{-5}$  a  $2^{15}$  (uniforme nos expoentes)
- gamma:  $2^{-15}$  a  $2^3$  (uniforme nos expoentes)
- epsilon: 0.05 a 1.0 (uniforme no intervalo)

```
[2]: # Defines
exp_min_C = -5
exp_max_C = 15
exp_min_gamma = -15
exp_max_gamma = 3
min_epsilon = 0.05
max_epsilon = 1.0
```

## 1.2 Carregamento dos dados

```
[3]: X = np.load("X.npy")
     y = np.load("y.npy")
```

## 1.3 Medida de erro:

Utilizamos como medida do erro a raiz quadrada do erro-médio , do inglês RSME.

## 1.4 Random Search

Escolha os valores para cada dimensão aleatoriamente, segundo uma distribuição

No nosso caso, teremos um grid exponencial equivale a uma distribuição uniforme no expoente.

```
[ ]: param_distributions = {'C': loguniform(2**exp_min_C, 2**exp_max_C),
                           'gamma': loguniform(2**exp_min_gamma, 2**exp_max_gamma) ,
                           'epsilon': uniform(min_epsilon, max_epsilon)}

start = time.time()
# Realiza a busca aleatória
# Se cv = None, utiliza 5 fold cross validation
random_search = RandomizedSearchCV(
    SVR(),
    param_distributions,
    random_state = 0,
    n_iter=125,
    scoring = 'neg_mean_squared_error',
    cv = None,
    refit = True,
    verbose=2)
random_search.fit(X, y)
time_elapsed = time.time() - start

[5]: print("Tempo de execução Random Search: {:.3f} s".format(time_elapsed))
     print("Melhores parâmetros: ", random_search.best_params_)
     regressor = SVR(kernel = 'rbf', C = random_search.best_params_['C'],
                     gamma = random_search.best_params_['gamma'],
                     epsilon = random_search.best_params_['epsilon'])

     scores = cross_val_score(regressor, X, y, scoring = 'neg_mean_squared_error',
                               cv=5)
     rmse = np.sqrt(np.mean(np.absolute(scores)))
     print("RMSE:", rmse)
```

Tempo de execução Random Search: 33.768 s

Melhores parâmetros: {'C': 12090.831718005036, 'epsilon': 0.7544144019235328, 'gamma': 4.540153912140535e-05}

RMSE: 4.100037405923898

## 1.5 Grid Search

Fazemos um grid search com grid de 5x5x5 selecionando uniformemente 5 expoentes de C e gamma e 5 valores de epsilon, com um grid exponencial linear nos expoentes.

```
[ ]: # C_values = [2**float(expo) for expo in np.random.uniform(low = exp_min_C,
    ↪ high = exp_max_C, size = 5)]
# gamma_values = [2**float(expo) for expo in np.random.uniform(low =
    ↪ exp_min_gamma, high = exp_max_gamma, size = 5)]
param_grid = {'C': loguniform.rvs(2**exp_min_C, 2**exp_max_C, size = 5),
    'gamma': loguniform.rvs(2**exp_min_gamma, 2**exp_max_gamma, size=
    ↪ 5) ,
    'epsilon': uniform.rvs(min_epsilon, max_epsilon, size= 5)}
print(param_grid)
strt = time.time()
# Realiza o Grid Search utilizando os parametros definidos em param_grid
grid_search = GridSearchCV(SVR(),
    param_grid,
    cv = None,
    scoring = 'neg_mean_squared_error',
    refit = True,
    verbose = 2)
grid_search.fit(X, y)
time_elapsed = time.time() - start
```

```
[7]: print("Tempo de execução Grid Search: {:.3f} s".format(time_elapsed))
print("Melhores parâmetros: ", grid_search.best_params_)
regressor = SVR(kernel = 'rbf', C = grid_search.best_params_['C'],
    gamma = grid_search.best_params_['gamma'],
    epsilon = grid_search.best_params_['epsilon'])

scores = cross_val_score(regressor, X, y, scoring = 'neg_mean_squared_error',
    ↪ cv=5)
rmse = np.sqrt(np.mean(np.absolute(scores)))
print("RMSE:", rmse)
```

Tempo de execução Grid Search: 83.842 s

Melhores parâmetros: {'C': 1602.7190592494144, 'epsilon': 0.8354722980853039,  
'gamma': 0.00028875853933915967}

RMSE: 4.834165296326991

## 1.6 Bayesian Search

A otimização bayesiana utilizando a bibliotec hyperopt requer a definição de uma função objetivo a ser minimizada. Como queremos maximizar a accuracy, utilizamos loss = -acc para ser minimizada.

```
[8]: def objective_func(search_space):
    C = search_space['C']
```

```

gamma = search_space['gamma']
epsilon = search_space['epsilon']
regressor = SVR(**{'C': 2**C, 'gamma': 2**gamma, 'epsilon': epsilon})

acc = cross_val_score(regressor, X, y, scoring='neg_mean_squared_error',
    ↪cv=None).mean()
    return {'loss': -acc, 'status': STATUS_OK}

bayesian_search_space = {
    'C': hp.uniform('C', exp_min_C, exp_max_C),
    'gamma': hp.uniform('gamma', exp_min_gamma, exp_max_gamma),
    'epsilon': hp.uniform('epsilon', min_epsilon, max_epsilon)
}
start = time.time()
best_params = fmin(objective_func, bayesian_search_space, algo=tpe.suggest,
    ↪max_evals=125)
time_elapsed = time.time() - start
best_parameters = {
    'C': 2**best_params['C'],
    'gamma': 2**best_params['gamma'],
    'epsilon': best_params['epsilon']
}

```

```

100%|      | 125/125 [01:52<00:00, 1.11trial/s, best loss:
15.153091976316869]

```

```

[9]: print("Tempo de execução Bayesian Search: {:.3f} s".format(time_elapsed))
    print("Melhores parâmetros:", best_parameters)
    regressor = SVR(C=best_parameters['C'], gamma=best_parameters['gamma'],
    ↪epsilon=best_parameters['epsilon'])
    scores = cross_val_score(regressor, X, y, scoring = 'neg_mean_squared_error',
    ↪cv=5)
    rmse = np.sqrt(np.mean(np.absolute(scores)))
    print("RMSE:", rmse)

```

```

Tempo de execução Bayesian Search: 112.572 s
Melhores parâmetros: {'C': 25339.857447098122, 'gamma': 3.1320582855738275e-05,
'epsilon': 0.5825810101914355}
RMSE: 3.892697262351244

```

## 1.7 PSO

há N partículas explorando o espaço das variáveis

cada partícula i inicia numa posição aleatória  $x_i$  e com uma “velocidade” (modulo e direção) aleatória  $V_i$

o grupo se lembra da melhor posição explorada até agora (g)

cada partícula tem um grupo de “amigos” ou vizinhos

o novo ponto da partícula é  $x_{i+1} = x_i + V_i$

a nova velocidade é atualizada para cada dimensão  $V_{i+1,d}$  e leva em consideração a velocidade anterior do ponto (“momento”), a posição do melhor ponto encontrado até agora (“componente cognitivo”) e o melhor ponto encontrado pelos seus vizinhos  $p_i$  (“componente social”)

```
[10]: def funcao(param):
    svr = SVR(kernel='rbf', C=param[0], gamma=param[1], epsilon=param[2])
    rmse = make_scorer(mean_squared_error, squared=False,
    ↪greater_is_better=False)
    scores = cross_val_score(svr, X, y, cv=5,
    ↪scoring=rmse) # 'neg_root_mean_squared_error'
    rmse_mean = -np.mean(scores)
    return rmse_mean

lb = np.array([2**exp_min_C, 2**exp_min_gamma, min_epsilon])
ub = np.array([2**exp_max_C, 2**exp_max_gamma, max_epsilon])

start = time.time()
xopt, fopt = pso(funcao, lb, ub, swarmsize=11, maxiter=11)
time_elapsed = time.time() - start

C_opt = str(xopt[0])
gamma_opt = str(xopt[1])
epsilon_opt = str(xopt[2])
print("Tempo de execução PSO: {:.3f} s".format(time_elapsed))
print("Melhores parâmetros: 'C': {0}, 'gamma': {1}, 'epsilon': {2}".
    ↪format(C_opt,
    ↪gamma_opt,
    ↪epsilon_opt))
print("RMSE: ", str(fopt))
```

Stopping search: maximum iterations reached --> 11

Tempo de execução PSO: 402.050 s

Melhores parâmetros: 'C': 21817.631831468443, 'gamma': 3.0517578125e-05,  
'epsilon': 0.05

RMSE: 3.716425672535194

## 1.8 Simulated annealing

mantem um ponto (estado)  $x$ ;

verifica um estado “vizinho”  $y$  (pode ser um ponto aleatório no espaço todo ou apenas em volta do  $x$ );

se  $f(y)$  é menor do que  $f(x)$  então  $y$  é o novo estado;

senão aceita  $y$  como o novo estado com probabilidade  $P(f(y)-f(x),T)$ , onde  $T$  é a "temperatura";  
 uma função comum para  $P$  é  $P = e^{\{-(f(y)-f(x))/T\}}$   $P = e^{\{-(f(y)-f(x))/T\}}$ ;

$T$  alta: aumenta a probabilidade de aceitar o ponto. Se  $T$  é infinito  $P = e^0 = 1$ ;

$T$  baixa: diminui a probabilidade. Com  $T=0$  a probabilidade é 0;

$T$  começa alta e vai diminuindo com o tempo - cooling schedule;

exploration no começo, e exploitation no final;

```
[16]: class SimAnn(Annealer):

    def move(self):
        self.state[0] = 2**np.random.uniform(low = exp_min_C, high = exp_max_C)
        self.state[1] = 2**np.random.uniform(low = exp_min_gamma, high = exp_max_gamma)
        self.state[2] = np.random.uniform(min_epsilon, max_epsilon)

    def energy(self):
        svr = SVR(kernel='rbf', C=self.state[0], gamma=self.state[1], epsilon=self.state[2])
        rmse = make_scorer(mean_squared_error, squared=False, greater_is_better=False)
        scores = cross_val_score(svr, X, y, cv=5, scoring=rmse)
        rmse_mean = -np.mean(scores)

        return rmse_mean

C = 2**np.random.uniform(low = exp_min_C, high = exp_max_C)
gamma = 2**np.random.uniform(low = exp_min_gamma, high = exp_max_gamma)
epsilon = np.random.uniform(min_epsilon, max_epsilon)
init_state = [C, gamma, epsilon]
sa = SimAnn(init_state)
sa.steps = 125
start = time.time()
best_params, rmse = sa.anneal()
time_elapsed = time.time() - start
print("Tempo de execução Simulated Annealing: {:.3f} s".format(time_elapsed))
print("Melhores parâmetros: 'C': {0}, 'gamma': {1}, 'epsilon': {2}".format(best_params[0],
best_params[1],
best_params[2]))
print('RMSE: ' + str(rmse))
```

Temperature	Energy	Accept	Improve	Elapsed	Remaining
-------------	--------	--------	---------	---------	-----------

2.50000            8.13    100.00%       0.00%       0:00:35       0:00:00

Tempo de execução Simulated Annealing: 34.821 s

Melhores parâmetros: 'C': 17644.155559829433, 'gamma': 4.8541623897673706e-05,  
'epsilon': 0.8190117619976423

RMSE: 4.304367131249085

## 1.9 CMA-ES

A implementação do algoritmo de otimização CMA-ES foi feita via biblioteca `optunza`. A implementação é encapsulada em um sampler default fornecido pela própria biblioteca, assim não sendo necessário a implementação de uma classe sampler específica, somente a função objetivo.

Utilize 125 chamadas da função.

```
[ ]: def objective(trial):
    c = trial.suggest_loguniform('c', 2**exp_min_C, 2**exp_max_C)
    gamma = trial.suggest_loguniform('gamma', 2**exp_min_gamma,
    ↪ 2**exp_max_gamma)
    epsilon = trial.suggest_uniform('epsilon', min_epsilon, max_epsilon)

    svr = SVR(kernel='rbf', C=c, gamma=gamma, epsilon=epsilon)
    scores = cross_val_score(svr, X, y, scoring = 'neg_mean_squared_error',
    ↪ cv=5)
    rmse = np.sqrt(np.mean(np.absolute(scores)))
    return rmse

sampler = optuna.samplers.CmaEsSampler()
cmaes = optuna.create_study(sampler=sampler)
start = time.time()
cmaes.optimize(objective, n_trials=125)
time_elapsed = time.time() - start

[18]: print("Tempo de execução CMA-ES: {:.3f} s".format(time_elapsed))
      print('Melhores parâmetros:', cmaes.best_params)
      print('RMSE:', cmaes.best_value)
```

Tempo de execução CMA-ES: 18.199 s

Melhores parâmetros: {'c': 2216.197086488184, 'gamma': 6.610965699443206e-05,  
'epsilon': 0.3284615721209558}

RMSE: 4.2324142272666485