MO431A_Tarefa4

May 2, 2021

MO431A - Fundamentos de Álgebra Linear e Otimização para Aprendizado de Máquina
Equipe:

- Maria Fernanda Tejada Begazo - RA 197488
- Jose Italo da Costa Silva - RA 265682
- Gian Franco Joel Condori Luna - RA 234826

Tarefa 04
A tarefa foi desenvolvida na linguagem python. Para isso utilizou-se notebooks jupyter no ambiente Google Colaboratory (Google Colab).
Pré-requisitos (codificação inicial, instalação e importação de pacotes):

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[ ]:
```
!pip install scikit-optimize
!pip install pyswarm
!pip install optuna
```

[ ]:
```
#Imports necessários:
import numpy as np
import cv2
from google.colab.patches import cv2_imshow
import time
import io
import random
from numpy import linalg
import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib.colors import LogNorm
from sympy import Derivative, diff, simplify
from sympy import Symbol, exp, Heaviside
from scipy.stats import loguniform

from sklearn.svm import SVR
from sklearn.model_selection import KFold
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV
from sklearn.model_selection import cross_val_score
from skopt import BayesSearchCV
from sklearn import metrics
#import tpyspark

#from hyperopt import fmin, tpe, hp. SparkTrials, STATUS_OK, Trials
from pyswarm import pso
import optuna

seed_ = 100
```

Objetivo

Vamos fazer a busca dos melhores hiperparametros para uma SVM para Regressão num banco de dados em particular. "X.npy" são os dados de entrada, e "y.npy" são os valores de saída corespondentes.

SVM Regressor

O "sklearn.svm.SVR" do skelearn implementa o regressor SVM e tem vários hiperparametros. Vamos usar o kernel rbf. Neste caso, há 3 hiperparametros que se considera como os mais importantes: C, gamma, e epsilon.

Vmos a fazer a busca no range:

- C: entre 2^{-5} e 2^{15} (uniforme nos expoentes)
- gamma: entre 2^{-15} e 2^{3} (uniforme nos expoentes)
- epsilon: entre 0.05 a 1.0 (uniforme neste intervalo)

Medida de erro

5-fold cross-validation e o processo de dividr o conjunto de dados em 5 conjuntos de tamanhos iguais, e treinar o regressor em 4 desses conjuntos e testar no conjunto restante. Mede-se então a média do desempenho do regressor nas 5 rodadas.

Para cada conjunto de hiperparametros, use 5-fold cross-validation para computar a média da raiz quadra do erro quadrado medio (RMSE) do SVM.

$$RMSE = \sqrt{\frac{\sum_{i=0}^{N}(x_{i,predicted} - x_{i,real})^2}{N}}$$

[ ]:
```
def SVM_kfolder(Xtra, Xtes, Ytra, Ytes, gamma, C, epsilon):
  errorT = 0
  for i in range(len(Xtra)):
    Xtrain = np.array(Xtra[i])
    Ytrain = np.array(Ytra[i])
    Xtest  = np.array(Xtes[i])
    Ytest  = np.array(Ytes[i])

    svr = SVR(kernel='rbf', gamma = gamma, epsilon=epsilon, C=C)
    svr.fit(Xtrain, Ytrain)

    pred = svr.predict(Xtest)
    error = 0.0
    for j in range(len(Xtest)):
      error += (pred[j] - Ytest[j])**2
    error = np.sqrt(error/ len(Xtest))

    errorT += error

  return errorT/len(Xtra)
```

1 Funções básica

[ ]:
```
X = np.load("/content/drive/My Drive/Algebra Lineal/X.npy")
Y = np.load("/content/drive/My Drive/Algebra Lineal/y.npy")

print("X: ", X.shape)
print("Y: ", Y.shape)
```

```
X:  (506, 13)
Y:  (506,)
```

[ ]:
```
def foldCrossValidation(kfold, X, Y):
  Idx_Xtrain = []
  Idx_Xtest = []
  kf = KFold(n_splits=kfold)
  for train, test in kf.split(X):
    Idx_Xtrain.append(train)
    Idx_Xtest.append(test)

  Xtrain = []
  Ytrain = []
  Xtest = []
  Ytest = []
  for i in range(len(Idx_Xtrain)):
    aux = []
    aux1= []
    for j in Idx_Xtrain[i]:
      aux.append(X[j])
      aux1.append(Y[j])
    Xtrain.append(list(aux))
    Ytrain.append(list(aux1))

    aux = []
    aux1= []
    for j in Idx_Xtest[i]:
      aux.append(X[j])
      aux1.append(Y[j])
    Xtest.append(list(aux))
    Ytest.append(list(aux1))
  return Xtrain, Xtest, Ytrain, Ytest
```

[ ]:
```
Xtrain, Xtest, Ytrain, Ytest = foldCrossValidation(5, X, Y)
```

2 Random Search

[ ]:
```
n_combinations = 125
np.random.seed(seed_)

C_range = np.random.uniform(-5, 15, n_combinations).astype(float)
C_range = 2**C_range

gamma_range = np.random.uniform(-15, 3, n_combinations).astype(float)
gamma_range = 2**gamma_range

epsilon_range = np.random.uniform(0.05, 1.0, n_combinations).astype(float)

hyperparameters = {'gamma':list(gamma_range), 'C':list(C_range), 'epsilon':
 ↪list(epsilon_range)}
```

[ ]:
```
def randomSearch(hyperparameters, Xdata, Ydata):
  randomCV = RandomizedSearchCV(SVR(kernel='rbf'),␣
 ↪param_distributions=hyperparameters, cv=5, n_iter=125,␣
 ↪scoring='neg_root_mean_squared_error', random_state = seed_)

  randomCV.fit(Xdata, Ydata)

  best_gamma   = randomCV.best_params_['gamma']
  best_C       = randomCV.best_params_['C']
  best_epsilon = randomCV.best_params_['epsilon']
  best_score   = randomCV.best_score_
  print("The best performing C value is: {:5.2f}".format(best_C))
  print("The best performing gamma value is: {:5.6f}".format(best_gamma))
  print("The best performing epsilon value is: {:5.6f}".format(best_epsilon))
  #print("The best score value is: {:5.6f}".format(best_score))

  return best_gamma, best_C, best_epsilon
```

[ ]:
```
gamma, C, epsilon = randomSearch(hyperparameters, X, Y)
error = SVM_kfolder(Xtrain, Xtest, Ytrain, Ytest, gamma, C, epsilon)
print("RMSE: {:5.6f}".format(error))
```

```
The best performing C value is: 25152.55
The best performing gamma value is: 0.000048
The best performing epsilon value is: 0.505709
RMSE: 4.400056
```

3 Grid Search

[ ]:
```
#O grid é 5x5x5
n_combinations = 5
np.random.seed(seed_)

C_grid = random.sample(list(C_range), k=n_combinations)
gamma_grid = random.sample(list(gamma_range), k=n_combinations)
epsilon_grid = random.sample(list(epsilon_range), k=n_combinations)

hyperparametersGrid = {'gamma':list(gamma_grid), 'C':list(C_grid), 'epsilon':
 ↪list(epsilon_grid)}
```

[ ]:
```
#cv = cross-validation
def gridSearch(hyperparameters, Xdata, Ydata):
  gridCV = GridSearchCV(SVR(kernel='rbf'), param_grid=hyperparameters, cv=5,␣
 ↪scoring='neg_root_mean_squared_error')

  gridCV.fit(Xdata, Ydata)

  best_gamma   = gridCV.best_params_['gamma']
  best_C       = gridCV.best_params_['C']
  best_epsilon = gridCV.best_params_['epsilon']

  print("The best performing C value is: {:5.2f}".format(best_C))
  print("The best performing gamma value is: {:5.6f}".format(best_gamma))
  print("The best performing epsilon value is: {:5.6f}".format(best_epsilon))

  return best_gamma, best_C, best_epsilon
```

[ ]:
```
gamma, C, epsilon = gridSearch(hyperparametersGrid, X, Y)
error = SVM_kfolder(Xtrain, Xtest, Ytrain, Ytest, gamma, C, epsilon)
print("RMSE: {:5.6f}".format(error))
```

```
The best performing C value is: 2116.92
The best performing gamma value is: 0.000236
The best performing epsilon value is: 0.980414
RMSE: 4.726221
```

4 Otimização Bayesiana

[ ]:
```
# log-uniform: understand as search over p = exp(x) by varying x
opt = BayesSearchCV( SVR(kernel='rbf'), hyperparameters, n_iter=125, cv=5,␣
 ↪scoring='neg_root_mean_squared_error')
opt.fit(X, Y)

best_gamma   = opt.best_params_['gamma']
```

```
best_C       = opt.best_params_['C']
best_epsilon = opt.best_params_['epsilon']

print("The best performing C value is: {:5.2f}".format(best_C))
print("The best performing gamma value is: {:5.6f}".format(best_gamma))
print("The best performing epsilon value is: {:5.6f}".format(best_epsilon))

error = SVM_kfolder(Xtrain, Xtest, Ytrain, Ytest, best_gamma, best_C,␣
 ↪best_epsilon)
print("RMSE: {:5.6f}".format(error))
```

```
/usr/local/lib/python3.7/dist-packages/skopt/optimizer/optimizer.py:449:
UserWarning:

The objective has been evaluated at this point before.


The best performing C value is: 7172.36
The best performing gamma value is: 0.000044
The best performing epsilon value is: 0.399274
RMSE: 3.906011
```

5 PSO

[ ]:
```
lb = np.array([-5, -15, 0.05])
ub = np.array([15, 3,     1.0])

def svr_fun(X):
  c = X[0]
  g = X[1]
  e = X[2]

  error = SVM_kfolder(Xtrain, Xtest, Ytrain, Ytest, 2**g, 2**c, e)
  return error

x_opt, y_opt = pso(svr_fun, lb, ub, swarmsize=11, maxiter=11)

print("C optimal: ", 2**x_opt[0])
print("Gamma optimal: ", 2**x_opt[1])
print("Epsilon optimal: ", x_opt[2])
print("RMSE: ", y_opt)
```

```
Stopping search: maximum iterations reached --> 11
C optimal:  4332.7722594521165
Gamma optimal:  5.321762164967722e-05
Epsilon optimal:  0.9800563470266999
RMSE:  4.052849838613556
```

6 Simulated Annealing

[ ]:
```
class SimulatedAnnelalingSampler(optuna.samplers.BaseSampler):
  def __init__(self, temperature=100):
    self._rng = np.random.RandomState(seed_)
    self._temperature = temperature #Temperatura Atual
    self._current_trial = None #Estado atual

  def sample_relative(self, study, trial, search_space):
    if search_space == {}:
      return {}

    #Calcular a probabilidade de transição
    prev_trial = study.trials[-2]
    probability  = 0.0

    if self._current_trial is None or prev_trial.value <= self._current_trial.
↪value:
      probability = 1.0
    else:
      probability = np.exp((self._current_trial.value - prev_trial.value) /␣
↪self._temperature)

    self._temperature *= 0.9 #Decrease temperatura

    #Transição do estado atual se o resultado previo é aceptado
    if self._rng.uniform(0, 1) < probability:
      self._current_trial = prev_trial

    params = {}
    for param_name, param_distribution in search_space.items():
      if not isinstance(param_distribution, optuna.distributions.
↪UniformDistribution):
        raise NotImplementedError('ONly suggest_uniform() is supported')

      current_value = self._current_trial.params[param_name]
      width = (param_distribution.high - param_distribution.low) * 0.1
      neighbor_low = max(current_value - width, param_distribution.low)
      neighbor_high = min(current_value + width, param_distribution.high)
      params[param_name] = self._rng.uniform(neighbor_low, neighbor_high)

    return params

  def infer_relative_search_space(self, study, trial):
    return optuna.samplers.intersection_search_space(study)

  def sample_independent(self, study, trial, param_name, param_distribution):
    independent_sampler = optuna.samplers.RandomSampler()
    return independent_sampler.sample_independent(study, trial, param_name,␣
 ↪param_distribution)
```

[ ]:
```
def objective(trial):
  c = trial.suggest_uniform('c', -5/20, 15/20)
  gamma = trial.suggest_uniform('gamma', -15/20, 3/20)
  epsilon = trial.suggest_uniform('epsilon', 0.05, 1.0)

  error = SVM_kfolder(Xtrain, Xtest, Ytrain, Ytest, 2**(20*gamma), 2**(20*c),␣
 ↪epsilon)

  return error

sampler = SimulatedAnnelalingSampler(1000)
study = optuna.create_study(sampler = sampler)
study.optimize(objective, n_trials=125)
```

[ ]:
```
param = study.best_params
print("C optimal: ", 2**(20*param['c']))
print("Gamma optimal: ", 2**(20*param['gamma']))
print("Epsilon optimal: ", param['epsilon'])
print("RMSE: ", study.best_value)
```

```
C optimal:  15123.181787068177
Gamma optimal:  3.19464329995032e-05
Epsilon optimal:  0.6084584631227788
RMSE:  3.8004961393230055
```

7 CMA-ES

[ ]:
```
sampler = optuna.samplers.CmaEsSampler(seed=seed_)
studyCMA = optuna.create_study(sampler=sampler)
studyCMA.optimize(objective, n_trials=125)
```

[ ]:
```
param = studyCMA.best_params
print("C optimal: ", 2**(20*param['c']))
print("Gamma optimal: ", 2**(20*param['gamma']))
print("Epsilon optimal: ", param['epsilon'])
print("RMSE: ", studyCMA.best_value)
```

```
C optimal:  13552.412596731656
Gamma optimal:  3.360577176909772e-05
Epsilon optimal:  0.433301864955961
RMSE:  3.7842910138176777
```

Algoritmos	C	log2(C)	gamma	log2(gamma)	epsilon	RMSE
Random Search	25152.55	14.61	0.000048	-14.35	0.505709	4.400056
Grid Search	2116.92	11.05	0.000236	-12.05	0.980414	4.726221
Optimização Bayesiana	7172.36	12.81	0.000044	-14.47	0.399274	3.906011
PSO	4332.77	12.08	0.000053	-14.20	0.980056	4.052850
Simulated Annealing	15123.19	13.88	0.000032	-14.93	0.608458	3.800496
CMA-ES	13552.41	13.73	0.000036	-14.76	0.433302	3.784291

8 Resultados

Nós observamos que para ter dados consistentes devemos instanciar uma semente. Assim vai se obter uma resposta similar em qualquer instancia de tempo. Ademais observamos que a Optimização Bayesiana da um RMSE baixo, mas demora muito tempo para descobrir os hiperparametros. Por outro lado, na tabela de abaixo, observamos que o melhor algoritmo foi CMA-ES observando o RMSE.

Por último, se observamos só a parte do log2(C) e log2(gamma), podemos deduzir que o melhor exponente é 14 e -15 respetivamente. E um epislon próximo a 0.43.