Supervised Learning – a4rl

André Filipe Frade Guerra

Novembro/2022

Conteúdo

Exercício 1

No exercício 1 começamos por inicializar as variáveis com os valores necessários para correr o algoritmo:

```
4   runs = 30
5   array = [[0,0],[0,1],[1,0],[1,1]]
6   dOR = [0,1,1,1]
7   dAND = [0,0,0,1]
8   alfa = [0.00010, 0.0010, 0.10, 1, 4]
```

De seguida, o inicio do algoritmo que começa por repetir o algoritmo tendo em conta o numero de alfas no array da variavél e o numero de ciclos pretendido, neste caso 30, atribuindo sempre novos valores aleatorios aos w's:

```
10  averagePerAlfa = []
11  standardDevPerAlfa = []
12  for w in range(len(alfa)):
13      numEpochsPerRun = []
14      for j in range(runs):
15          wOR0 = random.uniform(0, 0.9)
16          wOR1 = random.uniform(0, 0.9)
17          wOR2 = random.uniform(0, 0.9)
18          twOR0=0
19          twOR1=0
20          twOR2=0
21
22          wAND0 = random.uniform(0, 0.9)
23          wAND1 = random.uniform(0, 0.9)
24          wAND2 = random.uniform(0, 0.9)
25          twAND0=0
26          twAND1=0
27          twAND2=0
28
29          eOR = 1
30          eAND = 1
31          count = 0
```

Inicia-se o algoritmo, que vai atualizar os valores w's em cada interação, até que o resultado do erro seja 0:

```
32          while (eOR > 0.0 or eAND > 0.0):
33              resOR = []
34              resAND = []
35              for i in range(4):
36                  auxOR = wOR0 + wOR1 * array[i][0] + wOR2 * array[i][1]
37                  resOR.append(GetFuncValue(auxOR))
38
39                  auxAND = wAND0 + wAND1 * array[i][0] + wAND2 * array[i][1]
40                  resAND.append(GetFuncValue(auxAND))
41
42                  twOR0 = twOR0 + alfa[w] * (dOR[i] - GetFuncValue(auxOR))
43                  twOR1 = twOR1 + alfa[w] * array[i][0] * (dOR[i] - GetFuncValue(auxOR))
44                  twOR2 = twOR2 + alfa[w] * array[i][1] * (dOR[i] - GetFuncValue(auxOR))
45                  wOR0 = wOR0 + twOR0
46                  wOR1 = wOR1 + twOR1
47                  wOR2 = wOR2 + twOR2
48
49                  twAND0 = twAND0 + alfa[w] * (dAND[i] - GetFuncValue(auxAND))
50                  twAND1 = twAND1 + alfa[w] * array[i][0] * (dAND[i] - GetFuncValue(auxAND))
51                  twAND2 = twAND2 + alfa[w] * array[i][1] * (dAND[i] - GetFuncValue(auxAND))
52                  wAND0 = wAND0 + twAND0
53                  wAND1 = wAND1 + twAND1
54                  wAND2 = wAND2 + twAND2
55
56              eOR = DiffErro(resOR, dOR)
57              eAND = DiffErro(resAND, dAND)
58              count += 1
59
60          numEpochsPerRun.append(count)
```

Por fim podemos visualizar o resultado do exercício:

```
66          print("average number epochs: " + str(soma / len(numEpochsPerRun)))
67          print("standard deviation => " + str(Stdev(numEpochsPerRun)))
68          averagePerAlfa.append(soma / len(numEpochsPerRun))
69          standardDevPerAlfa.append(Stdev(numEpochsPerRun))
70
71      soma = 0
72      for j in averagePerAlfa:
73          soma += j
74      print("average number per alfa: " + str(soma / len(averagePerAlfa)))
75      print("standard deviation per alfa => " + str(Stdev(standardDevPerAlfa)))
```

```
average number epochs: 139.7
standard deviation => 35.3403923766182
average number epochs: 44.6
standard deviation => 10.812338630780424
average number epochs: 6.2
standard deviation => 1.939071942966532
average number epochs: 5.166666666666667
standard deviation => 0.6368324391514268
average number epochs: 4.0
standard deviation => 0.0
average number per alfa: 39.93333333333332
standard deviation per alfa => 13.379981869723638
```

Podemos concluir que quando o alfa aumenta de valor, o número de ciclos para chegar ao resultado é inferior.

Exercício 2

No exercício 2 começamos por inicializar as variáveis com os valores necessários para correr o algoritmo e ver o ficheiro iris.data que contem os dados a serem utilizados no algoritmo, de seguida, o inicio do algoritmo que começa por repetir o algoritmo tendo em conta o numero de “k”’s no array da variavél:

```
numRuns = 10
k = [3, 4, 7, 11]

ReadFile(setosa, versicolor, virginica, dataSet)

for j in k:
    Evaluate(j, j, dataSet, numRuns)
```

Neste algoritmo foi criada uma classe “Iris” que guarda os valores ao ler o ficheiro e a distância que é usada mais a frente pelo algoritmo:

```
class Iris:
  id = -1
  sepalLength = 0
  sepalWidth = 0
  petalLength = 0
  petalWidth = 0
  className = ""
  distance = 0
```

O método “ReadFile” lê o ficheiro e cria os objetos em arrays especificos, e tambem retorna 1 array com todos os objetos “dataSet”, array que é usado no algoritmo:

```
def ReadFile(setosa, versicolor, virginica, dataSet):
    f = open(os.getcwd() + "\\a4rl\iris.data", "r")
    w = 1
    for x in f:
        if x.__contains__("setosa"):
            s = x.replace(",Iris-setosa\n","")
            i = Iris()
            i.id = w
            i.sepalLength = float(s.split(",")[0])
            i.sepalWidth = float(s.split(",")[1])
            i.petalLength = float(s.split(",")[2])
            i.petalWidth = float(s.split(",")[3])
            i.className = "setosa"
            setosa.append(i)

        elif x.__contains__("virginica"):
            s = x.replace(",Iris-virginica\n","")
            i = Iris()
            i.id = w
            i.sepalLength = float(s.split(",")[0])
            i.sepalWidth = float(s.split(",")[1])
            i.petalLength = float(s.split(",")[2])
            i.petalWidth = float(s.split(",")[3])
            i.className = "virginica"
            virginica.append(i)

        elif x.__contains__("versicolor"):
            s = x.replace(",Iris-versicolor\n","")
            i = Iris()
            i.id = w
            i.sepalLength = float(s.split(",")[0])
            i.sepalWidth = float(s.split(",")[1])
            i.petalLength = float(s.split(",")[2])
            i.petalWidth = float(s.split(",")[3])
            i.className = "versicolor"
            versicolor.append(i)

        w += 1
```

O algoritmo começa com o metodo Evaluate, que corre o numero de vezes que pretendemos, neste caso 10, e no fim mostra a média do numero de previsoes corretas:

```
 97  def Evaluate(K, k, items, numRuns):
 98      accuracy = 0
 99      correct = 0
100      for i in range(numRuns):
101          random.shuffle(items)
102          accuracyAux, correctAux = K_FoldValidation(K, k, items)
103          accuracy += accuracyAux
104          correct += correctAux
105
106      print("k: "+ str(k) +"\naccuracy: " + str(accuracy / float(numRuns)) + " \navg correct predictions: "
107            + str(correct / len(items)) + "\ncorrect predictions: " + str(correct))
```

O método “K_FoldValidation” separa os dados em 2 arrays, o de testes e o de treino, percorre todos os dados no array de teste que são “treinados” tendo em conta o array de treino, e calcula no fim a precisão, verificando se a classe do dado a ser treinado é igual ao encontrado pelo algoritmo:

```
77  def K_FoldValidation(K, k, Items):
78      correct = 0
79      random.shuffle(Items)
80      trainingSet = []
81      testSet = []
82      for s in range(len(Items)):
83          if s < round(len(Items) * 0.7):
84              trainingSet.append(Items[s])
85          else:
86              testSet.append(Items[s])
87
88      for item in testSet:
89          itemClass = item.className
90          className = Classify(item, k, trainingSet)
91          if className == itemClass:
92              correct += 1
93
94      accuracy = correct / float(len(Items))
95      return accuracy, correct
```

O método “Classify” para cada dado de teste a ser classificado, vai definir a classe a que pertence “versicolor”, “setosa” ou “virginica”, tendo em conta os dados de treino:

```
103 def Classify(itemX, k, dataSet):
104     neighbors = []
105     for item in dataSet:
106         distance = EuclideanDistance(itemX, item)
107         neighbors = UpdateNeighbors(neighbors, item, distance, k)
108
109     className = CalculateNeighborsClass(neighbors, k)
110     return className
```

O método “EuclideanDistance” calcula a distância euclidiana tendo em conta os 4 parâmetros de cada objeto:

```
112 def EuclideanDistance(x, y):
113     s = 0
114     s += math.pow(x.sepalLength - y.sepalLength, 2)
115     s += math.pow(x.petalLength - y.petalLength, 2)
116     s += math.pow(x.sepalWidth - y.sepalWidth, 2)
117     s += math.pow(x.petalWidth - y.petalWidth, 2)
118     return math.sqrt(s)
```

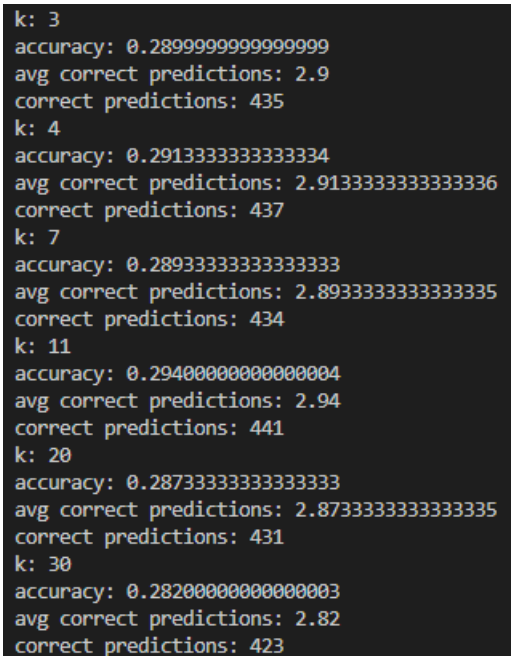
O método “UpdateNeighbors” escolhe o numero “k” ([3, 4, 7, 11]) de vizinhos mais próximos do ponto a ser avaliado, tendo em conta a distância anteriormente calculada:

```
120 def UpdateNeighbors(neighbors, item, distance, k):
121     if neighbors is None:
122         neighbors = []
123     if len(neighbors) < k:
124         item.distance = distance
125         neighbors.append(item)
126         neighbors.sort(key=lambda x: x.distance, reverse=False)
127     else:
128         if neighbors[-1].distance > distance:
129             item.distance = distance
130             neighbors[-1] = item
131             neighbors.sort(key=lambda x: x.distance, reverse=False)
132     return neighbors
```

Por fim o método “CalculateNeighborsClass”, retorna a classe a que o ponto em avaliação pode pertencer, tendo em conta a classe do maior número de vizinhos mais próximos, anteriormente calculado:

```
135 def CalculateNeighborsClass(neighbors, k):
136     countsetosa = 0
137     countversicolor = 0
138     countvirginica = 0
139     for i in range(round(k)):
140         if neighbors[i].className == "versicolor":
141             countversicolor += 1
142         elif neighbors[i].className == "setosa":
143             countsetosa += 1
144         elif neighbors[i].className == "virginica":
145             countvirginica += 1
146 
147     if countsetosa > countversicolor:
148         if countsetosa > countvirginica:
149             return "setosa"
150         else:
151             return countvirginica
152     elif countversicolor > countvirginica:
153         return "versicolor"
154     else:
155         return "virginica"
```

Ao correrm os algoritmo obtemos o seguinte resultado:



Assim, quando o valor de k é maior o número de previsões corretas é maior, até certo k. Podemos verificar que quando o k é 11, o algoritmo tem mais eficácia na previsão da classe.

Exercício 3

```
Entropy of  Petal Length  =  0.9580420222262995
```

Exercício 4

No exercício 4 foram utilizadas bibliotecas externas que implementam o algoritmo:

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB


numRuns = 10

X, y = load_iris(return_X_y=True)

for i in range(numRuns):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1)
    gnb = GaussianNB()
    y_pred = gnb.fit(X_train, y_train).predict(X_test)

    print("numero de previsões falhadas em %d = %d" % (X_test.shape[0], (y_test != y_pred).sum()))
```

```
numero de previsões falhadas em 45 = 3
numero de previsões falhadas em 45 = 3
numero de previsões falhadas em 45 = 3
numero de previsões falhadas em 45 = 3
numero de previsões falhadas em 45 = 3
numero de previsões falhadas em 45 = 3
numero de previsões falhadas em 45 = 3
numero de previsões falhadas em 45 = 3
numero de previsões falhadas em 45 = 3
numero de previsões falhadas em 45 = 3
```

Este algoritmo é mais consistente nos resultados obtendo sempre 3 falhas na previsão.