Distributed data analytics (2016-2017)
Information Systems and Machine Learning Lab
University of Hildesheim

Exercise Sheet 2

Mario Rodríguez Ruiz

April 28, 2017

Contents

List of Figures

List of Tables

1 Explain your system

	Home System
Machine	Asus Notebook ROG G60Jx
Operating System	Windows 10 Pro 64-bit
CPU	Intel Core i7 720QM @1.60GHz
Number of cores	4
Number of threads	8
RAM	16GB @665MHz (9-9-9-24)
Programming language version Python	v3.6.1:69c0db5 64 bit
Programming language version Java	v1.8

Table 1.1: My system

2 Formalize the task of k-NN

2.1 How the training and testing sets look like

Data set: There are **two types**, the training set and the testing set. To obtain these, the sample data are divided into two parts; one part is used as a **training set** to determine the classifier parameters and the other part, called a **testing set** (or set of generalization) is used to estimate the generalization error since the ultimate goal is for the classifier to get a (or overtraining), which consists of an overvaluation of the predictive capacity of the models obtained: in essence, it does not make sense to evaluate the quality of the model on the data that have served to construct it since this practice Leads to being over optimistic about their quality. Loss of generalization capacity leads to undesirable behavior (See Figure 2.1).

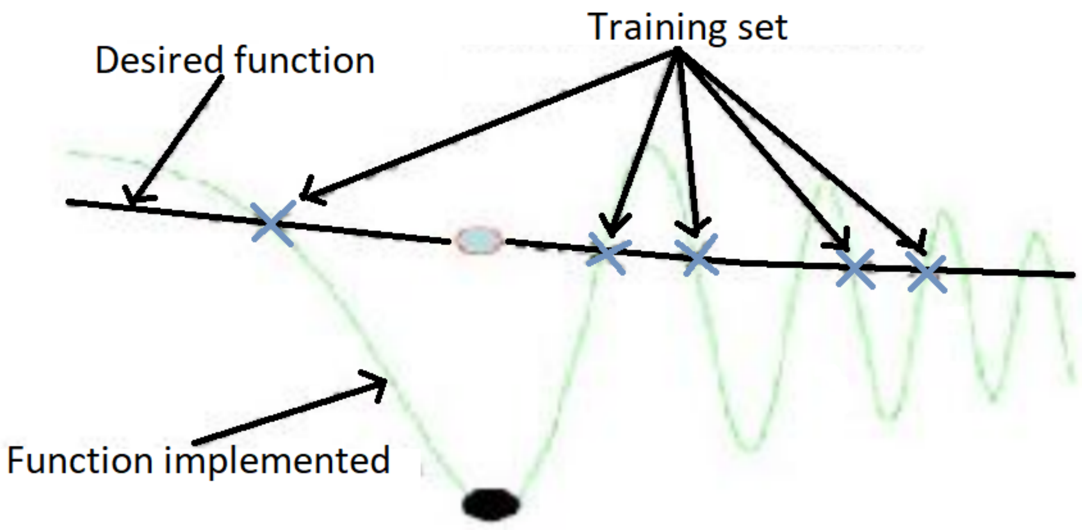


Figure 2.1: Loss of generalization capacity

The training set is usually divided into training sets (proper) and set of validation to fit the model (See Figure 2.2). 80% of the data are used to train the machine, 10% as a validation set and the remaining 10% to estimate the generalization (but it is only an orientative criterion).

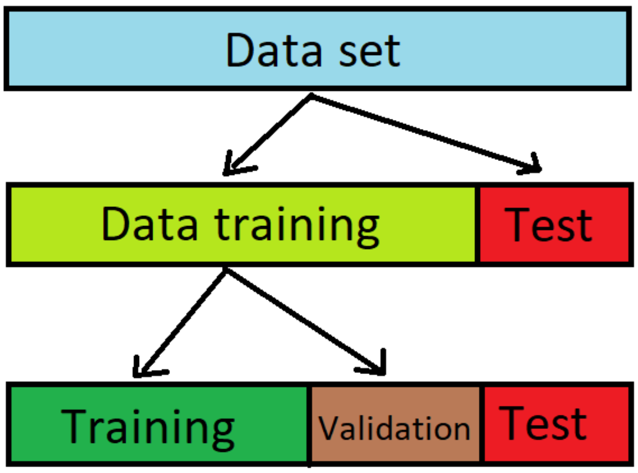


Figure 2.2: Data set division

One of the issues to consider in this algorithm is the choice of k. Depending on the value of k taken the effectiveness of the algorithm may vary to a greater or lesser extent. As shown in Figure 2.3, the value of k will determine whether the classification is correct or not. In addition, there is no way to know for what value of k we will get a greater precision in the results.

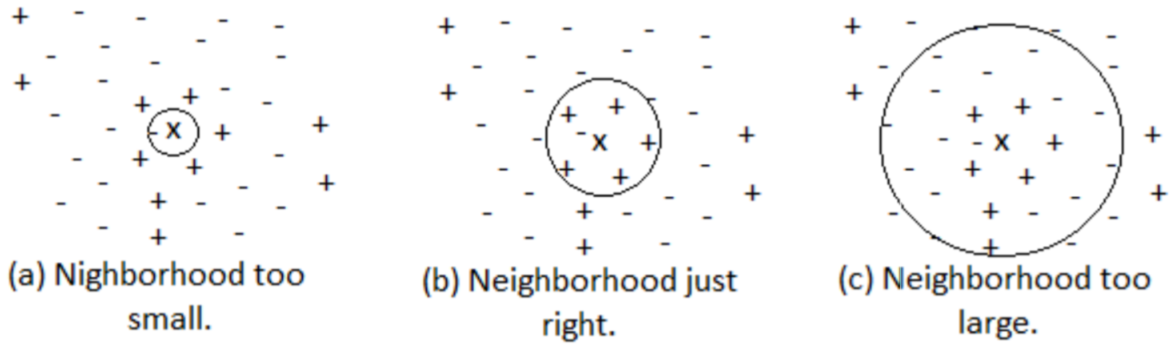

(a) Nighborhood too small.
(b) Neighborhood just right.
(c) Neighborhood too large.

Figure 2.3: The effectiveness of the algorithm

2.2 Distance measurement

The methods based on neighborhood are fundamentally dependent on the distance and, consequently, have their own characteristics such as proximity, remoteness and magnitude of length, among others.

In this case the **votes** of the k nearest neighbors will be weighted by their **cosine similarity (Figure 2.4)**.

$$sim(A,B) = \frac{\vec{A} \cdot \vec{B}}{|\vec{A}| \cdot |\vec{B}|}$$

Where the numerator represents the dot product of the vectors $\vec{A}$ and $\vec{B}$, while the denominator is the product of their Euclidean lengths.

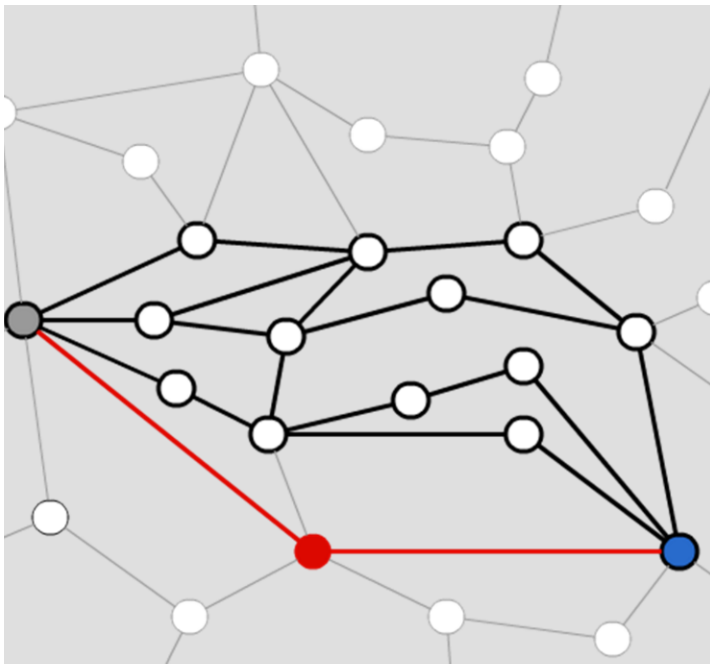


Figure 2.4: Cosine similarity distance

2.3 Pseudo-code of the task

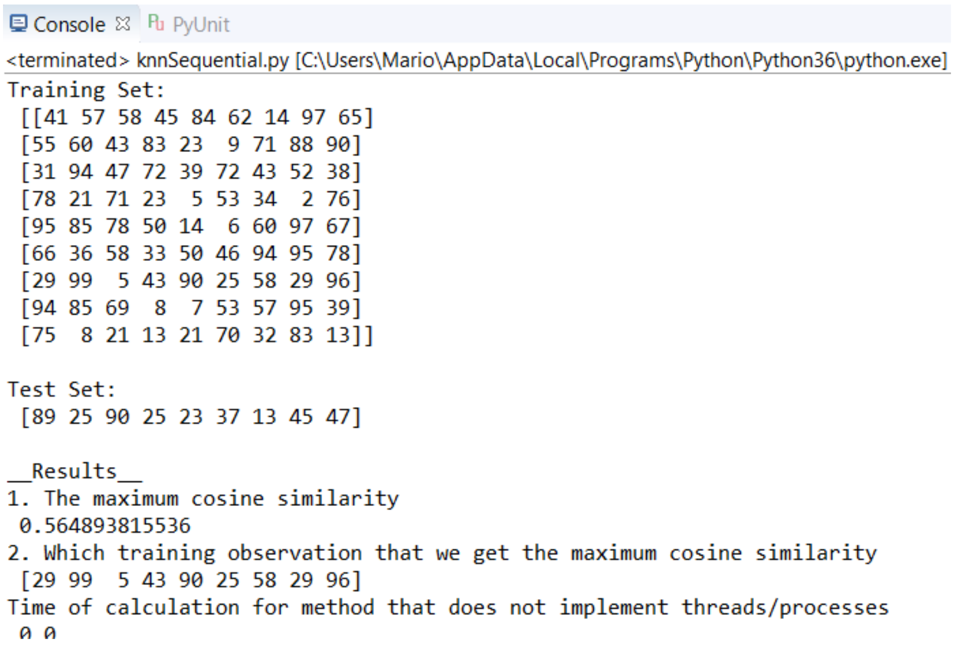
```
**START**
    **Input**: $D = \{(x_1, c_1), ..., (x_N, c_N)\}$
        $x = (x_1, ..., x_n)$ new case to classify
    **FOR** any object already **classified** $(x_i, c_i)$
        calculate $sim_i = sim_(x_i, x)$ cosine similarity distance
    Sort $sim_i(i = 1, ..., N)$ in ascending orden
    Stick to the **K** cases $D_x^K$ already classified closer to **x**
    Assign **x** to the most frequent class in $D_x^K$
**END**
```

3 Solve the k-NN sequentially

```python
def CosineSimilarity(A, B):
  sumX, sumXY, sumY = 0, 0, 0
  # For all size of A or B
  for i in range(len(A)):
    x = A[i]
    y = B[i]
    sumX += x*x
    sumY += y*y
    sumXY += x*y
  # return (A * B)/(|A|*|B|)
  return sumXY/math.sqrt(sumX*sumY)

def getNeighbors(trainingSet, testSet, k):
  distances = []
  for x in range(len(trainingSet)):
    dist = CosineSimilarity(testSet, trainingSet[x])
    distances.append((trainingSet[x], dist, x))
  distances.sort(key = operator.itemgetter(1))
  neighbors = []
  for x in range(k):
    neighbors.append(distances[x])
  return neighbors
```

An example of running the sequential knn can be seen in figure 3.1, that it shows the information that were requested for this exercise.



```
<terminated> knnSequential.py [C:\Users\Mario\AppData\Local\Programs\Python\Python36\python.exe]
Training Set:
 [[41 57 58 45 84 62 14 97 65]
 [55 60 43 83 23  9 71 88 90]
 [31 94 47 72 39 72 43 52 38]
 [78 21 71 23  5 53 34  2 76]
 [95 85 78 50 14  6 60 97 67]
 [66 36 58 33 50 46 94 95 78]
 [29 99  5 43 90 25 58 29 96]
 [94 85 69  8  7 53 57 95 39]
 [75  8 21 13 21 70 32 83 13]]

Test Set:
 [89 25 90 25 23 37 13 45 47]

__Results__
1. The maximum cosine similarity
 0.564893815536
2. Which training observation that we get the maximum cosine similarity
 [29 99  5 43 90 25 58 29 96]
Time of calculation for method that does not implement threads/processes
 0.0
```

Figure 3.1: Sequential k-NN execution example

4 Solve the k-NN in parallel

First, I have parallelized the part in which the workload is greatest, which is where the distances are calculated.

```python
def getNeighbors(trainingSet, testSet, k, proc):
  distances = []
  for x in range(len(trainingSet)):
    with Pool(proc) as pool:
      L = pool.starmap(CosineSimilarity, [(testSet,
          trainingSet[x])])
    dist = 1 - L[-1]
    distances.append((trainingSet[x], dist, x))
  distances.sort(key = operator.itemgetter(1))
  neighbors = []
  for x in range(k):
    neighbors.append(distances[x])
  return neighbors
```

However, the results with respect to time have not been satisfactory. The calculation has been done correctly, but the parallelization hasn't been optimal, which has resulted in the time has been increasing (Table 4.1).

Specification of test dimensions:
ROWS = 100
COLS = 100
k = 3

$testTraining[ROWS][COLS]$ randInt values
$testSet[COLS]$ randInt values

Cosine similarity MAX	Training obs. index	Time(s)	Processes
0.1917	3	3.4531	1
0.1917	3	4.5311	2
0.1917	3	5.7029	3
0.1917	3	5.7966	4
0.1917	3	5.9998	5
0.1917	3	7.0311	6
0.1917	3	7.8766	7
0.1917	3	8.6684	8

Table 4.1: First k-NN parallel

Secondly, I have implemented another new parallelization. This time the parallel part has been created before the call to the main function of the algorithm (**neighbors**).

```
for i in range(nt, NUM_TH+1):
  start = time.time()
  with Pool(i) as pool:
    L = pool.starmap(getNeighbors, [(trainingSet, testSet, k)
       ])
  end = time.time()
```

This time haven't improved the results with respect to time. However, executions with much larger sizes have been possible than in the other test without much delay of time (Table 4.2).

Specification of test dimensions:
ROWS = 10000
COLS = 10000
k = 3

$testTraining[ROWS][COLS]$ randInt values
$testSet[COLS]$ randInt values

Cosine similarity MAX	Training obs. index	Time	Processes
0.6956	8347	3.5012	1
0.6956	8347	3.6665	2
0.6956	8347	3.6371	3
0.6956	8347	3.5845	4
0.6956	8347	3.6355	5
0.6956	8347	3.7906	6
0.6956	8347	3.998	7
0.6956	8347	4.0914	8

Table 4.2: Second k-NN parallel