Data Mining and Machine Learning: Exercise 1

Johan Rodhe

September 9, 2016

1

We are given a set of data vectors $x_n = [x_1, x_2, ..., x_D]_n^T$ where $n = 1, 2, ..., N$. Assume a multivariate Gaussian model for the data.

a)

Q: Derive the maximum likelihood (ML) estimate for μ.

A: The log-likelihood function is given by:

$$lnp(X \mid \mu, \Sigma) = -\frac{ND}{2} ln(2\pi) - \frac{N}{2} ln|\Sigma| - \frac{1}{2} \sum_{n=1}^{N} (x_n - \mu)^T \Sigma^{-1} (x_n - \mu)$$

First I have to derive the function with respect to μ. The only part of the function that contains μ is the last one.

$$\frac{d}{d\mu} lnp(X \mid \mu, \Sigma) = \frac{d}{d\mu}(-\frac{1}{2} \sum_{n=1}^{N} (x_n - \mu)^T \Sigma^{-1} (x_n - \mu)) = -\frac{1}{2} \sum_{n=1}^{N} (2)(-1)(x_n - \mu)\Sigma^{-1} = \sum_{n=1}^{N} \Sigma^{-1}(x_n - \mu)$$

Setting this to zero we get:

$$\sum_{n=1}^{N} \Sigma^{-1}(x_n - \mu) \Leftrightarrow \sum_{n=1}^{N} (x_n - \mu) = 0 \Leftrightarrow \sum_{n=1}^{N} x_n - \sum_{n=1}^{N} \mu = 0 \Leftrightarrow \sum_{n=1}^{N} x_n = \sum_{n=1}^{N} \mu = N\mu \implies \mu_{ML} = \frac{1}{N} \sum_{n=1}^{N} x_n$$

b)

Q: Derive a sequential estimate for μ. In other words, assume that you have already obtained an ML estimate of μ based on the first N - 1 samples. Let's call that estimate μ_{ML}^{N-1}. Derive an expression for N -1 μ_{ML}^{N} based solely on μ_{ML}, the new sample x_N and N. A: From a) we have that $\mu_{ML}^{N} = \sum_{n=1}^{N} x_n$.

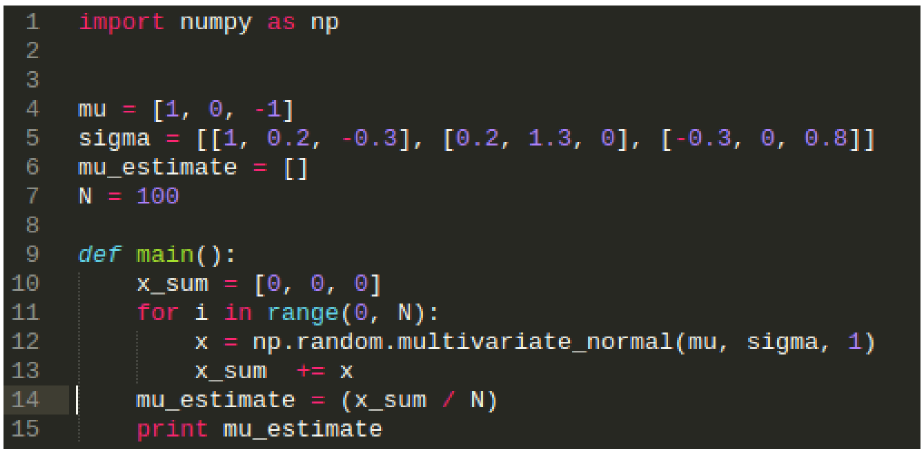
$$\sum_{n=1}^{N} x_n = \frac{1}{N} x_N + \frac{1}{N} \sum_{n=1}^{N-1} x_n = \frac{1}{N} x_N + \frac{N-1}{N} \mu_{ML}^{N-1} = \frac{1}{N} x_N + \frac{N}{N} \mu_{ML}^{N-1} - \frac{1}{n} \mu_{ML}^{N-1} = \mu_{ML}^{N-1} + \frac{1}{N}(x_N - \mu_{ML}^{N-1})$$

c)

Q: Generate $N = 100$ data points drawn from the distribution $\mathcal{N}(x \mid \mu, \Sigma)$ where

$$\mu = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}, \Sigma = \begin{bmatrix} 1 & 0.2 & -0.3 \\ 0.2 & 1.3 & 0 \\ -0.3 & 0 & 0.8 \end{bmatrix}$$

A: Below is a snippet of how i generated 100 data points from the given distribution in python.



```
1   import numpy as np
2
3
4   mu = [1, 0, -1]
5   sigma = [[1, 0.2, -0.3], [0.2, 1.3, 0], [-0.3, 0, 0.8]]
6   mu_estimate = []
7   N = 100
8
9   def main():
10      x_sum = [0, 0, 0]
11      for i in range(0, N):
12          x = np.random.multivariate_normal(mu, sigma, 1)
13          x_sum  += x
14      mu_estimate = (x_sum / N)
15      print mu_estimate
```

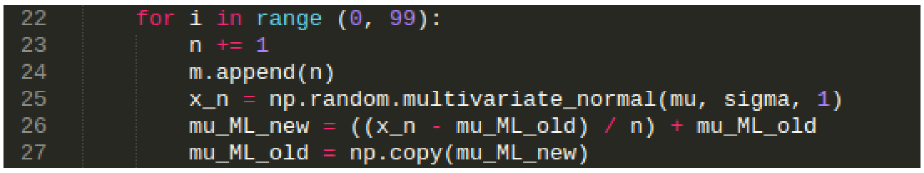
Figure 1: Python code sing the numpy library to generate data points. The formula from *a)* is used to calculate the batch estimate and can be seen on line 14.

Results:

$$\mu_{batchestimate} = \begin{bmatrix} 0.99978822 \\ -0.01694564 \\ -0.9024011 \end{bmatrix}$$

d)

Q: Initialize $\mu_{ML}^{N} = [0, 0, 0]^T$ and plot the sequential estimate μ_{ML}^{N} for $N = 1, 2, 3, ..., 100$.
A: Using the same library and distribution as in *c)* to generate the data points with one important difference. I don't generate all the data points and then perform the calculation for the estimate. Instead I now calculate a new estimate for each data point that is generated. See python code in figure below.



```
    for i in range (0, 99):
        n += 1
        m.append(n)
        x_n = np.random.multivariate_normal(mu, sigma, 1)
        mu_ML_new = ((x_n - mu_ML_old) / n) + mu_ML_old
        mu_ML_old = np.copy(mu_ML_new)
```

Figure 2: Python code to calculate the sequential estimate of μ_{ML}^{N}.

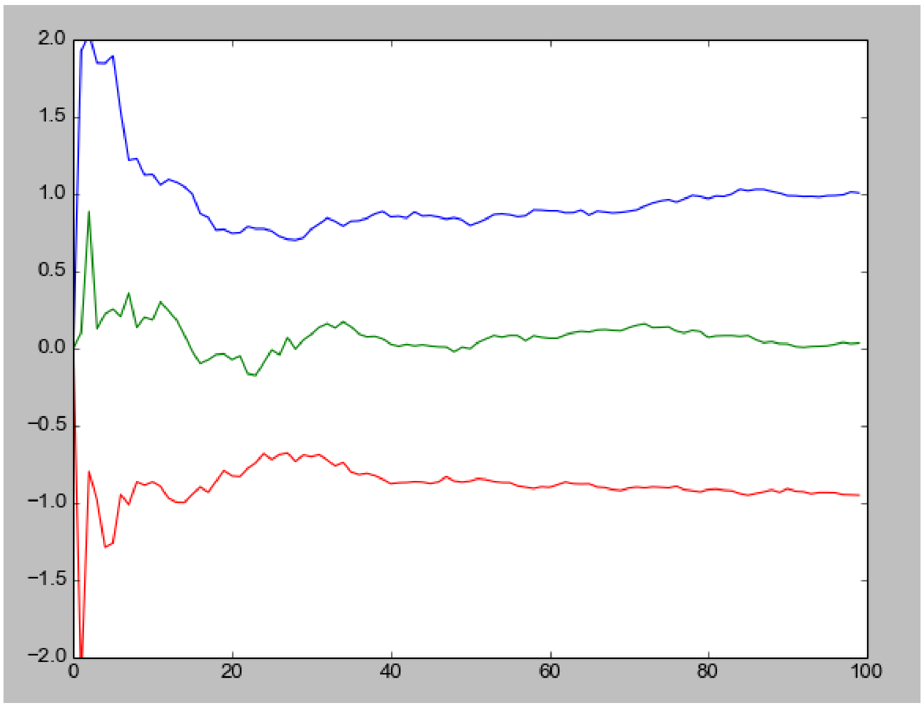


Figure 3: The sequential estimate for μ_{ML}^{N}.

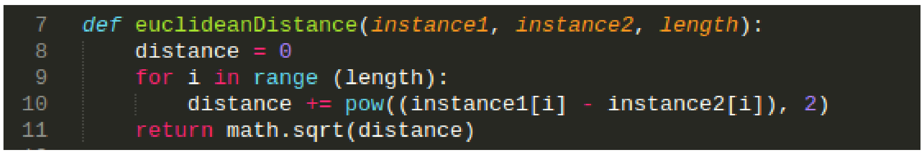
On line 26 we in Figure 2 we can see the formula derived in *b*).
Results: As can be seen in Figure 3, the red line converges to -1, the green line towards 0 and the blue towards 1. They all start in 0 because μ_{ML}^{N} is initialized as $[0, 0, 0]^T$.

2

a)

The kNN classifier is also implemented in python so the steps in the exercise were not precisely but still the algorithm is the same.

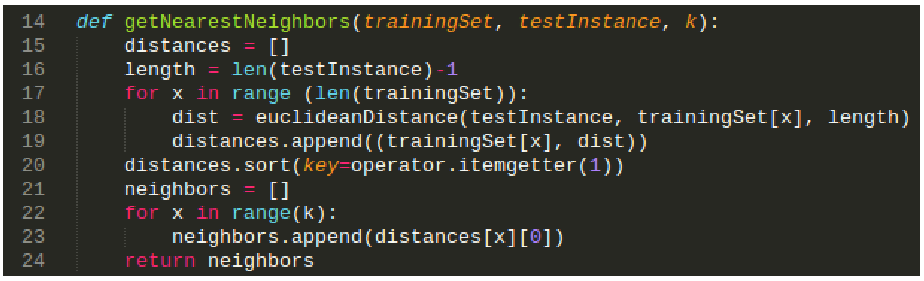
The points were generated in the same way as in 1*c*) and 1*d*) but now having two different distributions. They are then added to the training set and together with each point generated is added a 0 or a 1 as a label depending on what distribution it comes from. And then I can create another point using one of the two distributions as a test instance and verify that my program works. The euclidean distance is calculated with the function below:



```python
def euclideanDistance(instance1, instance2, length):
    distance = 0
    for i in range (length):
        distance += pow((instance1[i] - instance2[i]), 2)
    return math.sqrt(distance)
```

Figure 4: Python code to calculate the euclidean distance. Length is used to limit what attributes are calculated. Specifically so that the label is not considered.

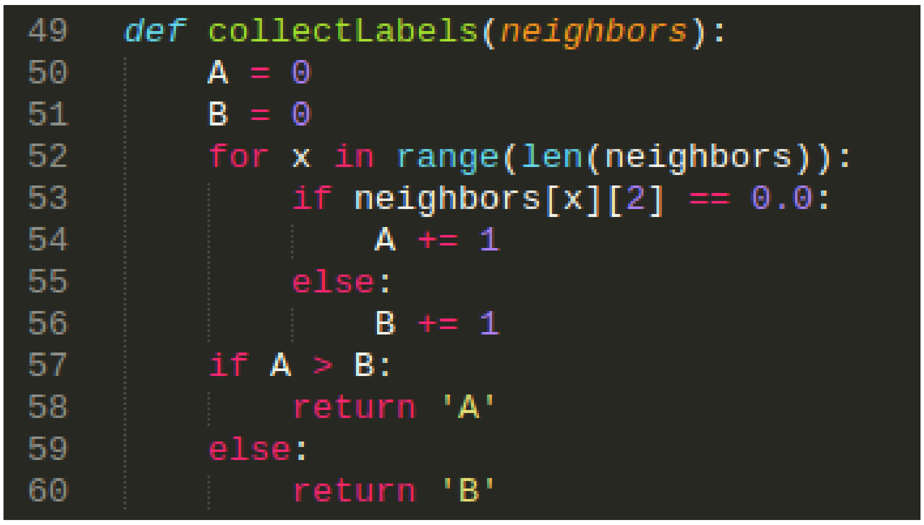
I create a data point from one of the two distributions and use it as input. The nearest neighbours is given by the function:



```python
def getNearestNeighbors(trainingSet, testInstance, k):
    distances = []
    length = len(testInstance)-1
    for x in range (len(trainingSet)):
        dist = euclideanDistance(testInstance, trainingSet[x], length)
        distances.append((trainingSet[x], dist))
    distances.sort(key=operator.itemgetter(1))
    neighbors = []
    for x in range(k):
        neighbors.append(distances[x][0])
    return neighbors
```

Figure 5: The distances are calculated and sorted. After that the k first ones from the sorted list are added to the neighbours.

The labels are then ”collected” with this function:



```python
def collectLabels(neighbors):
    A = 0
    B = 0
    for x in range(len(neighbors)):
        if neighbors[x][2] == 0.0:
            A += 1
        else:
            B += 1
    if A > B:
        return 'A'
    else:
        return 'B'
```

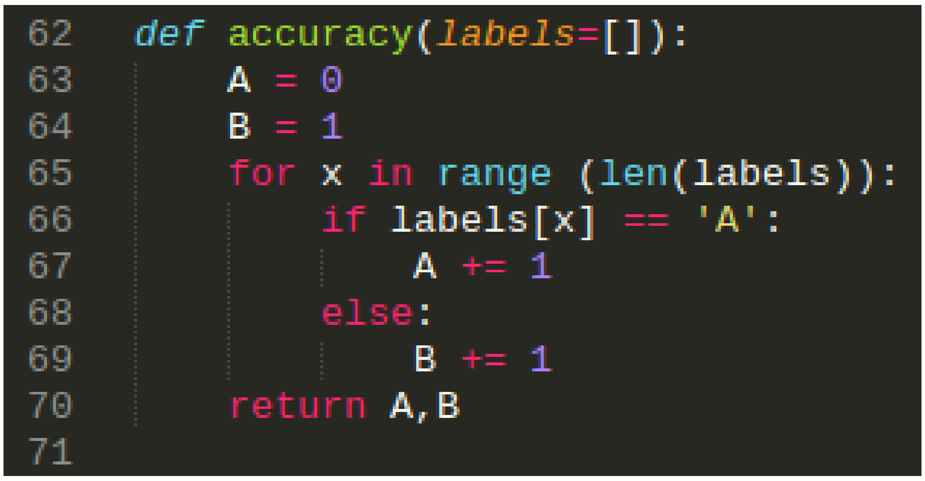
Figure 6: The list of neighbours is iterated and for each the label that occurs most often is returned.

Using a single input to test the algorithm I created a data point from the second distribution. If my algorithm is correct the program should give me the answer that the point belongs to B (the label corresponding to the second distribution).

The program does provide me with the correct label so so far I'm happy with my program.

b)

Now I create more data points and see how often my program gives me the wrong answer. The accuracy function gives me how many points that were assigned to each label.



```python
def accuracy(labels=[]):
    A = 0
    B = 1
    for x in range (len(labels)):
        if labels[x] == 'A':
            A += 1
        else:
            B += 1
    return A,B
```

Figure 7: The function goes through the list of labels and saves how many of each.

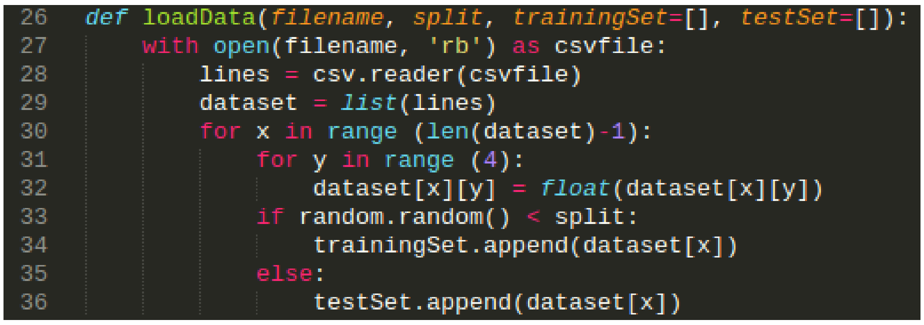
Creating 100 data points from the second distribution (that is with label B) i get the result: $(A, B) : (20, 80)$ (At most, repeating the process i get different results ranging from (5,95) to (20,80)). My error rate is $\frac{20}{100} = 80\%$.

c)

With different values of K i get similar results. The error rate does not differ any significant amount from what I can see. If k=1 the input is just assigned to the first neighbour. So with a higher k I should get a lower error rate. I cannot see the problem in my algorithm..

d)

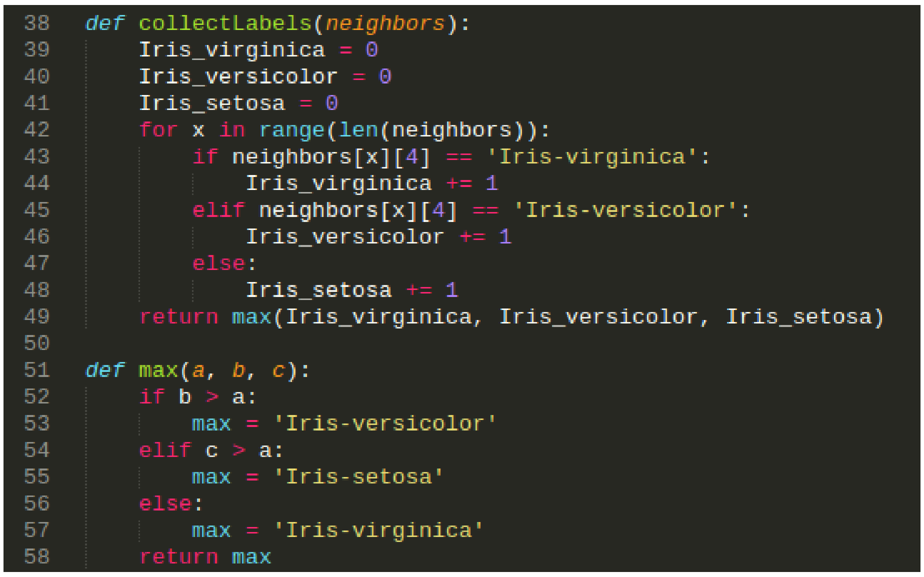
To test my algorithm on the Iris database I have to alter my program a little bit. For example I have to load the data. I added a function loadData:



```python
def loadData(filename, split, trainingSet=[], testSet=[]):
    with open(filename, 'rb') as csvfile:
        lines = csv.reader(csvfile)
        dataset = list(lines)
        for x in range (len(dataset)-1):
            for y in range (4):
                dataset[x][y] = float(dataset[x][y])
            if random.random() < split:
                trainingSet.append(dataset[x])
            else:
                testSet.append(dataset[x])
```

Figure 8: The loadData function loads the Iris data set into trainingSet in testSet. The split parameter decides how to split the data. I chose a ration 67/33. So 67% of the data set is the trainingSet and the rest is used as the test set.

I also altered the collectLabels function and the accuracy function:

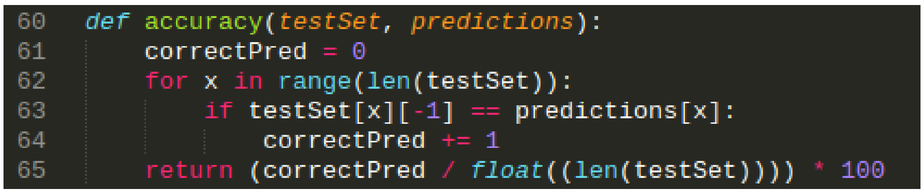


```python
def collectLabels(neighbors):
    Iris_virginica = 0
    Iris_versicolor = 0
    Iris_setosa = 0
    for x in range(len(neighbors)):
        if neighbors[x][4] == 'Iris-virginica':
            Iris_virginica += 1
        elif neighbors[x][4] == 'Iris-versicolor':
            Iris_versicolor += 1
        else:
            Iris_setosa += 1
    return max(Iris_virginica, Iris_versicolor, Iris_setosa)

def max(a, b, c):
    if b > a:
        max = 'Iris-versicolor'
    elif c > a:
        max = 'Iris-setosa'
    else:
        max = 'Iris-virginica'
    return max
```

Figure 9: The altered collectLabels function with a new max function added as well.

The accuracy function was altered as well:



```python
def accuracy(testSet, predictions):
    correctPred = 0
    for x in range(len(testSet)):
        if testSet[x][-1] == predictions[x]:
            correctPred += 1
    return (correctPred / float((len(testSet)))) * 100
```

Figure 10: The altered accuracy function. It now returns a percentage of wrong predictions.

Trying the program on the iris I get the following results:

k	misclassification rate
0	65%
1	9%
2	6%
3	5%
4	4.5%
5	3.5%
7	4%
9	4%

With higher values than 9 the result didn't change much.

3

An expert system approach based on principal component analysis and adaptive neuro-fuzzy inference system to diagnosis of diabetes disease

Authors: Kemal Polat∗, Salih Günes

Electrical and Electronics Engineering Department, Selcuk University, 42035 Konya, Turkey

Available online 20 October 2006

`<http://www.sciencedirect.com/science/article/pii/S1051200406001370>`

The study focuses on diabetes. Diabetes is in turn a disease that contributes to other diseases such as heart disease and increases the risk for developing kidney disease, blindness and nerve damage.

The dataset they use in the study was collected from UJI Repository of Machine Learning Databases, 1996,http://www.ics.uci.edu/ mlearn/MLRepository.html. The data set was picked from a larger set of data held by the National Institutes of Diabetes and Digestive and Kidney Diseases. The response variable is binary and take the value '0' and '1'. '0' for negative test for diabetes and '1' for positive test. The study used 8 different variables according to clinical findings.

- Number of times pregnant
- Plasma glucose concentration

- Diastolic blood pressure
- Triceps skin fold thickness (mm)
- 2-h serum insulin (mu U/ml)
- Body mass index
- Diabetes pedigree function
- Age (years)

The study suggests an expert system that has two stages. In the first step principal component analysis (PCA) was used to reduce the dimensions from 8 to 4. In the second step they used adaptive neuro-fuzzy inference system (ANFIS) classifier to diagnosis of diabetes disease.

There is a lot of information stored that could help with medical diagnosis. But the gap between how much information we have and how much of it we understand is widening. To try and shorten this gap data mining and different forms of expert systems can be used.

The study obtained a test classification accuracy of 89,47%. This is the highest classification accuracy among classifier report from literature.

The main result of the study is this successful rate. This leads to the conclusion that an expert system like the one used in the study is proposed for diagnosis of diabetes. It can provide some very useful assistance in decision-making in the medical field.