AI for Biotechnology

Exercise 2

Prof. Dr. Dominik Grimm
Bioinformatics Research Lab
TUM Campus Straubing for Biotechnology and Sustainability

Exercise E2.1

The euclidean distance between two points $\boldsymbol{x}^{[a]} \in \mathbb{R}^{1\times m}$ and $\boldsymbol{x}^{[b]} \in \mathbb{R}^{1\times m}$ is defined as follows:

$$d(\boldsymbol{x}^{[a]}, \boldsymbol{x}^{[b]}) = \sqrt{\sum_{i=1}^{m} \left(\boldsymbol{x}_i^{[a]} - \boldsymbol{x}_i^{[b]}\right)^2} \tag{1}$$

To implement Equation 1 in Python a `for` loop is needed. This could be computationally expensive when m is large (several thousand to millions of features). The dot-product between two arbitrary vectors $\boldsymbol{a} \in \mathbb{R}^{n\times 1}$ and $\boldsymbol{b} \in \mathbb{R}^{n\times 1}$ is defined as follows:

$$\boldsymbol{a}^\top \boldsymbol{b} = (a_1, \ldots, a_n) \cdot \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} = \sum_{i=1}^{n} a_i b_i \tag{2}$$

a) Show that you can reformulate Equation 1 as a dot product.

Solution:
Let $\boldsymbol{c} \in \mathbb{R}^{m\times 1}$ be a column-vector. Then we can write:

$$\boldsymbol{c} = \boldsymbol{x}^{[a]^\top} - \boldsymbol{x}^{[b]^\top}$$

Thus, the dot product $\boldsymbol{c}^\top \boldsymbol{c} = \left(\boldsymbol{x}^{[a]^\top} - \boldsymbol{x}^{[b]^\top}\right)^\top \left(\boldsymbol{x}^{[a]^\top} - \boldsymbol{x}^{[b]^\top}\right)$ is equivalent to $\sum_{i=1}^{m} \left(\boldsymbol{x}_i^{[a]} - \boldsymbol{x}_i^{[b]}\right)^2$.
Thus we can rewrite the Euclidean distance as:

$$d(\boldsymbol{x}^{[a]}, \boldsymbol{x}^{[b]}) = \sqrt{\boldsymbol{c}^\top \boldsymbol{c}}$$

This expression can be written in Python in a singe line and no `for` loop is needed.

b) Implement both versions in Python (`euclidean_distance_naive` and `euclidean_distance_dot`) and test your implementations with the following code:

```python
import numpy as np
import time

def euclidean_distance_naive(a,b):
    dist = 0
    for i in range(a.shape[1]):
        dist += (a[0,i] - b[0,i])**2
    dist = np.sqrt(dist)
    return dist

def euclidean_distance_dot(a,b):
```

```
    dist = np.sqrt(((a.T-b.T).T).dot(a.T-b.T))
    return dist[0,0]

#Simple Test with row vectors of size 1 x 2
a = np.array([[3,5]])
b = np.array([[6,9]])
print(euclidean_distance_naive(a,b))
print(euclidean_distance_dot(a,b))
print()

#Test with random row vectors of size 1 x 10 Million
#Random numbers with seed
np.random.seed(42)
m = 10**7 #10 Million Features
a = np.random.random((1,m))
b = np.random.random((1,m))

#stop time of computation
start = time.process_time()
#compute distance
print(euclidean_distance_naive(a,b))
delta = (time.process_time()-start)
print("Computation Time Naive: %f s" % delta)

start = time.process_time()
print(euclidean_distance_dot(a,b))
delta = (time.process_time()-start)
print("Computation Time Fast: %f s" % delta)
```

Exercise E2.2

We use the `scikit-learn` package to simulate some toy data as illustrated in Figure 1:

```
%matplotlib inline
import sklearn.datasets as datasets
import pylab as pl
#Simulate Toy Dataset
X, y = datasets.make_circles(n_samples=150, shuffle=True,
                             noise=0.2, random_state=42,
                             factor=0.1)
```

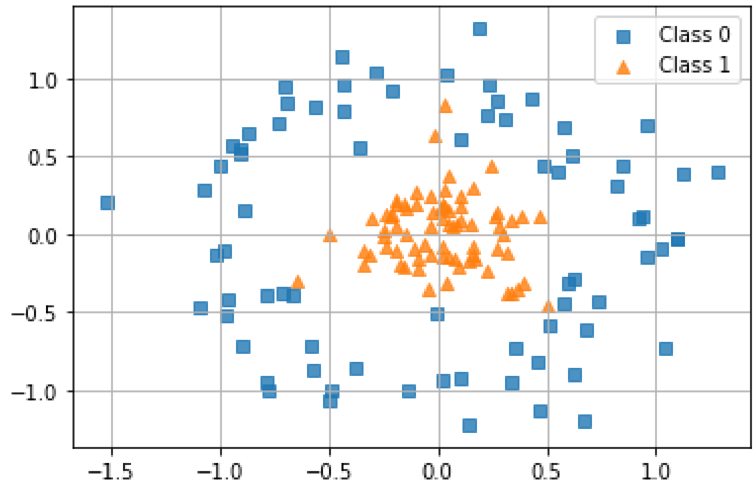

Figure 1: Toy data simulated with `scikit-learn`

a) Write a Python snippet to determine how many samples, features and classes the dataset has. Return also the number of samples for each of the classes (hint: have a look at the NumPy Method `unique`).

```
#Extract Information from Toy Dataset
print("Total #Sampels:\t\t" + str(X.shape[0]))
print("Number of Features:\t" + str(X.shape[1]))
labels, counts = np.unique(y, return_counts=True)
for i in range(len(labels)):
    print("#Samples class " + str(labels[i]) + ":\t" + str(counts[i]))
```

Output:
Total #Sampels: 150
Number of Features: 2
#Samples Class 0: 75
#Samples Class 1: 75

b) Reproduce the plot shown in Figure 1 using the `Matplotlib` library (hint: have a look at the `marker` argument of the scatter function using the matplotlib documentation).

```
#Plot the two features for each class
X_class0 = X[y==0,:] #Samples for class 0
X_class1 = X[y==1,:] #Samples for class 1
pl.scatter(X_class0[:,0],X_class0[:,1],
          alpha=0.8,label="Class 0",marker="s")
pl.scatter(X_class1[:,0],X_class1[:,1],
          alpha=0.8,label="Class 1",marker="^")
pl.grid()
pl.legend()
```

c) Add the two query points $q_1 = (-0.8, 0.3)$ and $q_2 = (-0.1, 0.1)$ to the scatter plot.

```
q1 = np.array([-0.8,0.3])
q2 = np.array([-0.1,0.1])
pl.scatter(q1[0],q1[1],alpha=1,label="Q1",marker="D")
pl.scatter(q2[0],q2[1],alpha=1,label="Q2",marker="P")
```

```
pl.grid()
pl.legend()
```

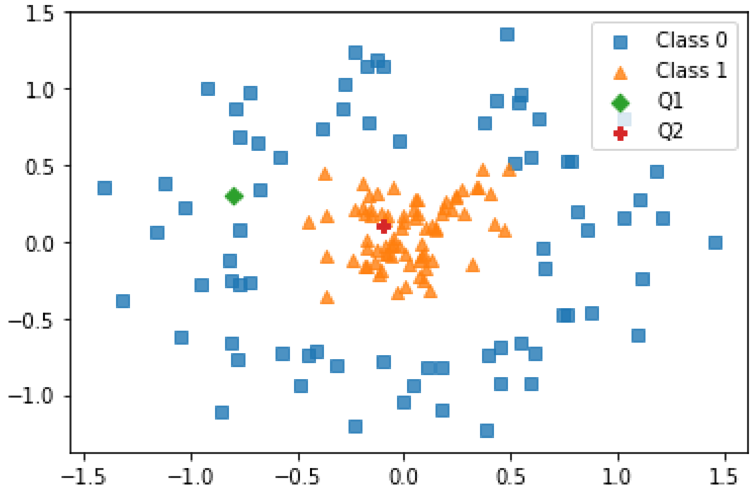

Figure 2: Toy data including query points q_1 and q_2

Exercise E2.3

Implement a function to classify a given query point q using the k-Nearest-Neighbor algorithm from scratch. Do not use the algorithms implemented in `scikit-learn`. The function should have the name `knn_predict` and should accept the arguments, `X`, `y`, `query_point` and `k`. Test your implemented function using the two query points q_1 and q_2 from the previous exercise for $k \in \{1, 10, 50, 100\}$. How are the query points classified using your implementation? (Hint: the `NumPy` functions `unique`, `argsort`, `argmax` and `norm` could be useful. However, there are several ways how to implement this function. You can also compare your implementation with the results from scikit-learn.)

```
#Implementation without for loops
def knn_predict(X,y,query_point,k=1):
    diff = X-query_point
    dist = np.linalg.norm(diff,axis=1)
    knn = np.argsort(dist)[:k]
    (label, counts) = np.unique(y[knn],return_counts=True)
    prediction = np.argmax(counts)
    return label[prediction]

#Alternative implementation
def knn_predict(X,y,query_point,k=1):
    distances = []
    #compute all distances
    for i in range(X.shape[0]):
        distance = euclidean_distance_dot(X[i,:],query_point)
        distances.append(distance)
    #sort distances in descending order
    indices_closest_points = np.argsort(distances)
    #use slicing to get the indices for the first k closest neighbors
    knn_indices = indices_closest_points[:k]
    #majority vote
    class0_count = (y[knn_indices]==0).sum()
```

```
    class1_count = (y[knn_indices]==1).sum()
    if class0_count>class1_count:
        return 0
    else:
        return 1
```

Output:

```
q1 = np.array([-0.8,0.3])
print("q1 for k1: ", knn_predict_fast(X,y,q1,k=1))
print("q1 for k10: ", knn_predict_fast(X,y,q1,k=10))
print("q1 for k50: ", knn_predict_fast(X,y,q1,k=50))
print("q1 for k100: ", knn_predict_fast(X,y,q1,k=100))
q2 = np.array([-0.1,0.1])
print("q2 for k1: ", knn_predict_fast(X,y,q2,k=1))
print("q2 for k10: ", knn_predict_fast(X,y,q2,k=10))
print("q2 for k50: ", knn_predict_fast(X,y,q2,k=50))
print("q2 for k100: ", knn_predict_fast(X,y,q2,k=100))
```

```
q1 for k1:  0
q1 for k10:  0
q1 for k50:  1
q1 for k100:  1
q2 for k1:  1
q2 for k10:  1
q2 for k50:  1
q2 for k100:  1
```