Abgabe Intelligente Adaptive Systeme Laborversuch 1

Malte Hoffmann

Tobias Schlauch

Steve Arnold Pouton Chedjou

V1A1

a)

```
def getKNearestNeighbors(x, X, k=1):  # realizes nearest neighbor search of x in database
    """
    compute the k nearest neighbors for a query vector x given a data matrix X
    :param x: the query vector x
    :param X: the N x D data matrix (in each row there is data vector) as a numpy array
    :param k: number of nearest-neighbors to be returned
    :return: return list of k line indixes referring to the k nearest neighbors of x in X
    """
    return np.argsort([np.sum(np.power((x - X[i]), 2)) for i in range(len(X))])
```

Für alle Datenvektoren d in X ziehen wir elementweise den Datenvektor x von d ab und quadrieren diese Differenz. Danach summieren wir alle Quadrate auf. Die daraus entstandene Liste sortieren wir mit argsort(), so dass wir den Index des kleinsten Elements an erster Stelle haben. Diese Liste geben wir zurück. Das kleinste Element hat hierbei den geringsten Abstand.

b)

```
Data matrix X=
 [[1 2 3]
 [2 3 4]
 [3 4 5]
 [4 5 6]]
Test vector x= [1.5 3.6 5.7]
Euklidean distances to x:  [10.100000000000001, 3.500000000000001, 2.9000000000000004, 8.299999999999999]
idx_knn= [2 1 3 0]
The k Nearest Neighbors of x are the following vectors:
The 1 th nearest neighbor is: X[ 2 ]= [3 4 5]  with distance  1.7029386365926402
The 2 th nearest neighbor is: X[ 1 ]= [2 3 4]  with distance  1.8708286933869709
```

Ausgabe unseres Codes von V1A1 a)

c)

sortieren ist n * log(n)
N := len(X), X = Datenmatrix -> Anzahl Datenvektoren => man braucht etwa O(1) Schritte mehr
D := len(x), x = Datenvektor -> Anzahl Vektorelemente => man braucht etwa O(N) Schritte mehr
k = Anzahl nearest neighbour => hat kaum Auswirkung auf die Laufzeit
Schnellere Verfahren sind z.B. KD-Trees

V1A2

a)
Klassen:
- Classifier, abstrakte Klasse, die das Grundgerüst und die zu implementierenden Funktionen angibt und eine eigene Funktion crossvalidate(self,S,X,T) implementiert
- KNNClassifier, konkrete Klasse, die von Classifier erbt und alle Funktionen implementiert und zusätzlich eine eigene Funktion getKNearestNeighbors(self, x, k, X) implementiert
- FastKNNClassifier, konkrete Klasse, die von KNNClassifier erbt und die fit(self,X,T) und getKNearestNeighbors(self, x, k=None) überschreibt. Diese müssen noch vervollständigt werden

Methoden:
- __init__(self,C) -> Kontrsuktor der Klasse, wobei C die Anzahl der zu unterscheidenden Klassen angibt
- fit(self,X,T) -> Funktion stellt sicher, dass die Datenmatrix ein zweidimensionales Array ist und der Datenvektor ein eindimensionales Array ist. Die Anzahl der Klassen wird in C gespeichert, allerdings nur wenn die Klassen durchgänig mit ganzen Integerzahlen durchnummeriert wurden
- predict(self, x) -> muss von abgeleitenten Klassen implementiert werden. Soll die Wahrscheinlichkeiten, dass der neue Datenvektor x zu einer der Klassen aus T gehört ausgeben
- crossvalidate(self,S,X,T) -> Teilt die Datenmenge in S Teile und trainiert das Model mit S-1 Teilen der Datenmenge, der S-Teil wird nach dem Trainieren zum Validieren benutzt. Am Ende wird die Wahrscheinlichkeit der Fehlklassifikation ausgegeben und eine Matrix in der angegeben ist mit welcher Wahrscheinlichkeit ein Objekt der Klasse j als ein Objekt der Klasse i klassifiziert wird.

b) Die Klasse "lernt" nicht wirklich. Es wäre ein bereits trainiertes Netz, dass einfach nur in seiner Matrix nachschaut welcher bisherige Datenvektor am nächsten des neuen Datenvektor ist.

```python
def getKNearestNeighbors(self, x, k=None, X=None):
    """
    compute the k nearest neighbors for a query vector x given a data matrix X
    :param x: the query vector x
    :param X: the N x D data matrix (in each row there is data vector) as a numpy array
    :param k: number of nearest-neighbors to be returned
    :return: list of k line indexes referring to the k nearest neighbors of x in X
    """

    if (k == None): k = self.k  # per default use stored k
    if (X == None): X = self.X  # per default use stored X

    return_list = np.argsort([np.sum(np.power((x - X[i]), 2)) for i in range(len(X))])[:k]
    return return_list  # REPLACE: Insert/adapt your code from V1A1_KNearestNeighborSearch.py
```

Code-Erklärung siehe V1A1 a). Zusätzlich schneiden wir die Liste nach dem k-Eintrag ab.

```python
def predict(self, x, k=None):
    """
    Implementation of classification algorithm, should be overwritten in any derived classes
    :param x: test data vector
    :param k: search k nearest neighbors (default self.k)
    :returns prediction: label of most likely class that test vector x belongs to
                         if there are two or more classes with maximum probability then one class is chosen randomly
    :returns pClassPosteriori: A-Posteriori probabilities, pClassPosteriori[i] is probability that x belongs to class i
    :returns idxKNN: indexes of the k nearest neighbors (ordered w.r.t. ascending distance)
    """
    if k == None: k = self.k                                    # use default parameter k?
    idxKNN = self.getKNearestNeighbors(x, k)                    # get indexes of k nearest neighbors of x
    pClassPosteriori = np.zeros(self.C)                         # initialise pClassPosteriori with zeros
    for i in idxKNN:                                            # go through every entry in idxKNN
        pClassPosteriori[self.T[i]] = pClassPosteriori[self.T[i]] + 1   # count how often the class is in idxKNN
    prediction = np.argsort(pClassPosteriori)[-1]               # the most propable class is the one with the most nearest neighbour
    pClassPosteriori = pClassPosteriori / k                     # divide through k
    return prediction, pClassPosteriori, idxKNN                 # return predicted class, a-posteriori-distribution, and indexes of nearest neighbors
```

Codeerklärung siehe Kommentare im Code

c)

```
Data matrix X=
 [[1 2 3]
 [2 3 4]
 [3 4 5]
 [4 5 6]]
Class labels T=
 [0 1 0 1]
Test vector x= [1.5 3.6 5.7]
Euklidean distances d= []

Classification with the naive KNN-classifier:
Test vector is most likely from class  0
A-Posteriori Class Distribution: prob(x is from class i)= [1. 0.]
Indexes of the k= 1  nearest neighbors: idx_knn= [2]

Classification with the naive KNN-classifier:
Test vector is most likely from class  1
A-Posteriori Class Distribution: prob(x is from class i)= [0.5 0.5]
Indexes of the k= 2  nearest neighbors: idx_knn= [2 1]

Classification with the naive KNN-classifier:
Test vector is most likely from class  1
A-Posteriori Class Distribution: prob(x is from class i)= [0.33333333 0.66666667]
Indexes of the k= 3  nearest neighbors: idx_knn= [2 1 3]

Classification with the fast KNN-classifier:
Test vector is most likely from class  1
A-Posteriori Class Distribution: prob(x is from class i)= [0.33333333 0.66666667]
Indexes of the k= 3  nearest neighbors: idx_knn= [2 1 3]

Process finished with exit code 0
```

d) Man sollte für 2 Klassen immer ein ungerades k angeben, damit ausgeschlossen wird, dass gleiche viele neighbour aus den Klassen zurückgeliefert wird (z.B. 1 Neighbour aus Klasse 0 und 1 Neighbour aus Klasse 1).

```
def fit(self, X, T):
    """
    Train classifier by creating a kd-tree
    :param X: Data matrix, contains in each row a data vector
    :param T: Vector of class labels, must have same length as X, each label should be integer in 0,1,...,C-1
    :returns: -
    """
    KNNClassifier.fit(self,X,T)              # call to parent class method (just store X and T)
    self.kdtree = spatial.KDTree(X)
```

Verwendung der KDTree-Funktion aus dem Modul spatial

```
def getKNearestNeighbors(self, x, k=None):  # realizes fast K-nearest-neighbor-search of x in data set X
    """
    fast computation of the k nearest neighbors for a query vector x given a data matrix X by using the KD-tree
    :param x: the query vector x
    :param k: number of nearest-neighbors to be returned
    :return idxNN: return list of k line indexes referring to the k nearest neighbors of x in X
    """
    if(k==None): k=self.k                       # do a K-NN search...
    Null, index = self.kdtree.query(x, k)       # query returns the index and distance, we save the distance in an
                                                # unused variable because we dont need it
    if(k == 1):                                 # check if k == 1 because then query returns only an int
        return [index]                          # return list if k == 1 because we iterate over it to get pClassPosteriori
    return index                                # if k =/= 1  query already returns an array
```

Codeerklärung siehe Kommentare im Code

```
Classification with the fast KNN-classifier:
Test vector is most likely from class  1
A-Posteriori Class Distribution: prob(x is from class i)= [0.33333333 0.66666667]
Indexes of the k= 3  nearest neighbors: idx_knn= [2 1 3]
```

Ausgabe unseres Programmes

V1A3

a)

- 0%, da alle neighbours (=1) von einer Klasse sind
- Nein, da es „Ausreiser“ geben kann. Also Datenvektoren die eigentliche im Bereich einer anderen Klasse liegen, dadurch kann es vorkommen, dass der Vektor mit dem kleinsten Abstand eine andere Klasse hat als alle Datenvektoren um ihn herum.
- Man kann mit neuen Daten testen und über die Fehlerwahrscheinlichkeit den Mittelwert bilden. Es kann die Matrix mit der Fehlprognose-Wsk pro Klasse gebildet werden.
- Kreuzvalidierung ist ein Verfahren zum Trainieren von neuronalen Netzten, dabei wird eine Datenmenge N in k-Teile aufgeteilt. Ein k-tel der Daten wird zur Validierung genutzt, der Rest zum Trainieren. Nach dem Training wird ein neues Model trainiert und ein anderer Teil der Daten wird zum Validieren benutzt. Hierdurch können alle Daten verwendet werden. Jedoch benötigt man viel Rechenpower/Zeit, da k Modelle trainiert werden müssen.

b)

-S = gibt an in wie viele Teile der Datensatz unterteilt werden soll
-perm = Liste mit zufälliger Reihenfolge der Zahlen 0 bis N-1, jede Zahl kommt nur einmal vor
-Xp = Liste mit Werten aus X, wobei die Sortierung von perm vorgegeben ist
-Tp = Liste mit den Labels für die Vektoren in Xp mit gleicher Sortierung
-idxS = Liste mit Indizes für die S-Teile der Größe N/S
-for idxTest in idxS = Sucht den Teil aus, der als Validierungsdaten genutzt wird, sodass idxTest die Indizes der Validierungsdaten enthält. Es wird mit for durch die gesamte Datenmenge iteriert
-X_learn/X_test = Indizes der Daten, die als Lerndaten verwendet werden (alle außer idxTest)
-T_learn /T_test = Indizes der Labels für X_learn, haben gleiche Sortierung
-"S = 1" = idxLearn=idxTest -> es wird der gesamte Datensatz als Lerndaten verwendet und danach als Testdaten
-for i in range(len(X_test)) = über alle Datenvektoren in den Lerndaten iterieren

-pClassError = addiert alle Fehler auf, wie oft ich insgesamt falsch lag
-pConfErrors = Eine Matrix, die angibt mit welcher Wsk eine echte Klasse einer Klasse zugewiesen wird, z.Bsp. wenn ich in 3 Fällen Klasse 1 als 0 und in 1 Fall Klasse 0 als 1 vorhersage [9 3]
[1 7]

c)
- Um alle Klassen, die wir in V1A2_Classifier definiert und implementiert haben in unserem neuen Skript verwenden zu können
- N1 und N2 sind die Datenvektoren mit denen gearbeitet wird. N1 ist von Klasse 0 und N2 von Klasse 1. N gibt die Anzahl der Spalten der Datenmatrix an, also wie viele Datenvektoren in der Matrix sind. D gibt die Dimensionalität der Datenvektoren an, bzw. wie viele Zeilen die Datenmatrix X hat.
- die Datenvektoren sind Gaußverteilt mit den Mittelwerten mu1 = [1,1] und mu2 = [3,1]. Die Kovarinazmatrizen sind sigma1 = [[1,0.5], [0.5,1]] und sigma2 = [[1,0.5], [0.5,1]]
- pE_naive ist die Wahrscheinlichkeit einer Fehl-Klassifikation
- pCE_naive ist die Matrix, die die Wahrscheinlichkeit eine Objekt der Klasse a einer Klasse b zuzuordnen, hierbei kann b auch a sein
- t_naive ist die geschätzte Dauer zum Lernen des gesamten Datensatzes

```
# (ii.b) test of KD-tree KNN classifier
print("\nFast KNN Classifier based on KD-Trees:","\n---------------------------------------")
fknnc = FastKNNClassifier(knnc)                # create an FastKNNClassifier object
t1 = time.perf_counter()                       # start time logging
pE_kdtree,pCE_kdtree=fknnc.crossvalidate(S,X,T) # "train" the model with crossvalidation
t2 = time.perf_counter()                       # end time logging
t_kdtree = t2-t1                               # calc how long crossvalidation took

print("S=", S, " fold Cross-Validation of kdTree ", k, "-NN-Classifier requires ", t_kdtree, " seconds. Confusion error probability matrix is \n", pCE_kdtree)
print("Probability of a classification error is pE = ", pE_kdtree)
```

d)
k = 1, S = 1

```
Data size: N= 1000 , D= 2

Naive KNN Classifier:
-----------------------
S= 1  fold Cross-Validation of naive  1 -NN-Classifier requires  34.2681732  seconds. Confusion error probability matrix is
 [[1. 0.]
 [0. 1.]]
Probability of a classification error is pE =  0.0
New data vector x_test= [2 1]  is most likely from class  1 ; class probabilities are p_class =  [0. 1.]
New data vector x_test= [5 1]  is most likely from class  1 ; class probabilities are p_class =  [0. 1.]
New data vector x_test= [-1  1]  is most likely from class  0 ; class probabilities are p_class =  [1. 0.]

Fast KNN Classifier based on KD-Trees:
---------------------------------------
S= 1  fold Cross-Validation of kdTree  1 -NN-Classifier requires  0.628719199999999  seconds. Confusion error probability matrix is
 [[1. 0.]
 [0. 1.]]
Probability of a classification error is pE =  0.0
```

k = 5, S = 2

```
Data size: N= 1000 , D= 2

Naive KNN Classifier:
-----------------------
S= 2  fold Cross-Validation of naive  5 -NN-Classifier requires  17.4545535  seconds. Confusion error probability matrix is
 [[0.85  0.164]
 [0.15  0.836]]
Probability of a classification error is pE =  0.157
New data vector x_test= [2 1]  is most likely from class  0 ; class probabilities are p_class =  [0.6 0.4]
New data vector x_test= [5 1]  is most likely from class  1 ; class probabilities are p_class =  [0. 1.]
New data vector x_test= [-1  1]  is most likely from class  0 ; class probabilities are p_class =  [1. 0.]

Fast KNN Classifier based on KD-Trees:
--------------------------------------
S= 2  fold Cross-Validation of kdTree  5 -NN-Classifier requires  0.8213492999999978  seconds. Confusion error probability matrix is
 [[0.822 0.186]
 [0.178 0.814]]
Probability of a classification error is pE =  0.182
```

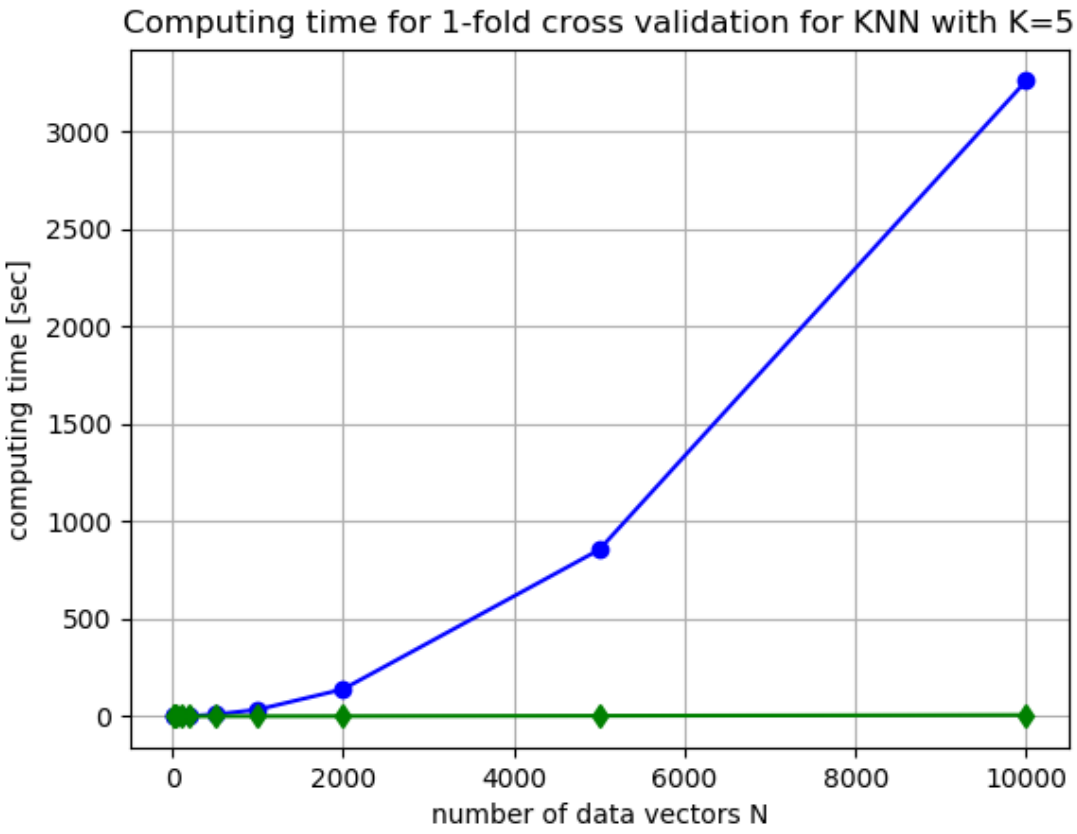
k = 11, S = 5

```
Data size: N= 1000 , D= 2

Naive KNN Classifier:
-----------------------
S= 5  fold Cross-Validation of naive  11 -NN-Classifier requires  26.746110199999997  seconds. Confusion error probability matrix is
 [[0.858 0.126]
 [0.142 0.874]]
Probability of a classification error is pE =  0.134
New data vector x_test= [2 1]  is most likely from class  1 ; class probabilities are p_class =  [0.27272727 0.72727273]
New data vector x_test= [5 1]  is most likely from class  1 ; class probabilities are p_class =  [0. 1.]
New data vector x_test= [-1  1]  is most likely from class  0 ; class probabilities are p_class =  [1. 0.]

Fast KNN Classifier based on KD-Trees:
--------------------------------------
S= 5  fold Cross-Validation of kdTree  11 -NN-Classifier requires  0.605733500000003  seconds. Confusion error probability matrix is
 [[0.784 0.186]
 [0.216 0.814]]
Probability of a classification error is pE =  0.201
```

K	Knnc	Fknnc
1	(2 1) = 1 (5 1) = 1 (-1 1) = 0	(2 1) = 1 (5 1) = 1 (-1 1) = 0
5	(2 1) = 0 (5 1) = 1 (-1 1) = 0	(2 1) = 0 (5 1) = 1 (-1 1) = 0
11	(2 1) = 1 (5 1) = 1 (-1 1) = 0	(2 1) = 1 (5 1) = 1 (-1 1) = 0
111	(2 1) = 0 (5 1) = 1 (-1 1) = 0	(2 1) = 0 (5 1) = 1 (-1 1) = 0
511	(2 1) = 0 (5 1) = 1 (-1 1) = 0	(2 1) = 0 (5 1) = 1 (-1 1) = 0

Die predicteten Klassen sind gleich, da der gleiche Algorithmus zur Berechnung angewandt wird. Allerdings ist die Fehlerwahrscheinlichkeit unterschiedlich, da bei der Kreuzvalidierung unterschiedliche Permutation erzeugt werden und damit die Modelle unterschiedlich „lernen“.

e)



N	T(N)knnc [sec]	T(N)fknnc [sec]
10	0.006057800000000224	0.003429399999999916
20	0.017594200000000004	0.010788999999999938
50	0.10557680000000014	0.024012499999999992
100	0.38096629999999987	0.050906600000000246
200	1.5312214000000002	0.11193839999999966
500	9.5174417	0.2974475999999999
1000	34.2212492	0.6367327999999972
2000	138.96248359999998	1.1612240000000043
5000	854.7705021	2.9269702999999936
10000	3261.0793586000004	6.712470999999823