

Abgabe SMD Blatt 10

von Nico Guth, David Venker, Jan Jäkel

Aufgabe 20 k-NN Klassifikation

a) Worauf müssen Sie bei einem k-NN-Algorithmus achten, wenn die Attribute sich stark in ihren Größenordnungen unterscheiden?

Wichtig ist der Abstand zu den k nächsten Nachbarn des Trainings Datensatzes.

Dafür kann ein beliebiges Abstandsmaß gewählt werden.

Allerdings sind so gut wie alle Abstandsmaße nicht für den Fall optimiert, dass die Dimensionen auf verschiedenen Größenordnungen liegen.

Also kann ein Attribut mit viel größeren Werten als die anderen, diese stark überwiegen und so wird quasi nur dieses Attribut zur Klassifizierung verwendet.

Um dieses Problem zu lösen sollte man die Daten skalieren. Z.B. auf standard-normal verteilte Werte.

b)

Warum bezeichnet man den k-NN als sogenannten "lazy learner"?

Es findet nicht wirklich ein Training statt. Beim Training wird nur der Trainings Datensatz abgespeichert.

Als "lazy learner" wird ein Maschinelles Lerner bezeichnet, dessen Modellbildung nicht während dem Training sondern während der Anwendung geschieht.

Wie sind die Laufzeiten für Lern- und Anwendungs-Phase?

<https://towardsdatascience.com/k-nearest-neighbors-computational-complexity-502d2c440d5>

Varibalen:

- n : Anzahl der Samples im Trainings-Datensatz
- d : Anzahl der Dimensionen/Attributen
- k : Anzahl der betrachteten Nachbarn

Komplexität:

Methode	Training Laufzeit	Anwendung Laufzeit	Training Speicher	Anwendung Speicher
---------	----------------------	-----------------------	----------------------	-----------------------

Methode	Training Laufzeit	Anwendung Laufzeit	Training Speicher	Anwendung Speicher
Bruteforce Methode	$O(1)$	$O(knd)$	$O(1)$	$O(1)$
$k - d$ -Baum Methode	$O(dn \log(n))$	$O(k \log(n))$	$O(dn)$	$O(1)$
Ball-Baum Methode	$O(dn \log(n))$	$O(k \log(n))$	$O(dn)$	$O(1)$

Also ist in der Bruteforce Methode quasi keine Laufzeit beim Training vorhanden, dafür aber ziemlich viel Laufzeit während der Anwendung. Bei den Baum Methoden ist die Laufzeit des Trainings höher als die der Anwendung, da normalerweise $d \cdot n > k$

Wie sind sie im Vergleich zu anderen Algorithmen, wie bspw. einem Random Forest?

<https://www.thekerneltrip.com/machine/learning/computational-complexity-learning-algorithms/>

t = Anzahl der Bäume

Komplexität des Random Forests:

- Training: $O(n^2dt)$
- Anwendung: $O(dt)$

Also dauert im normalfall das Training eines Random Forests deutlich länger als eines k-NN. Aber die Anwendung dauert beim k-NN deutlich länger als beim Random Forest.

Wenn man die Bruteforce Methode von k-NN benutzt, ist verbraucht das Training so gut wie

c) Implementieren Sie einen k-NN zur Klassifikation mit einer Struktur wie in `class_structure.py`

Implementiert in `knn.py` und weiter unten im PDF angehangen. (Bruteforce Methode)

Test mit `sklearn.datasets.make_blobs`

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
```

In [2]:

```
X, y = make_blobs(n_samples=1000, centers=10, n_features=5, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, rand
```

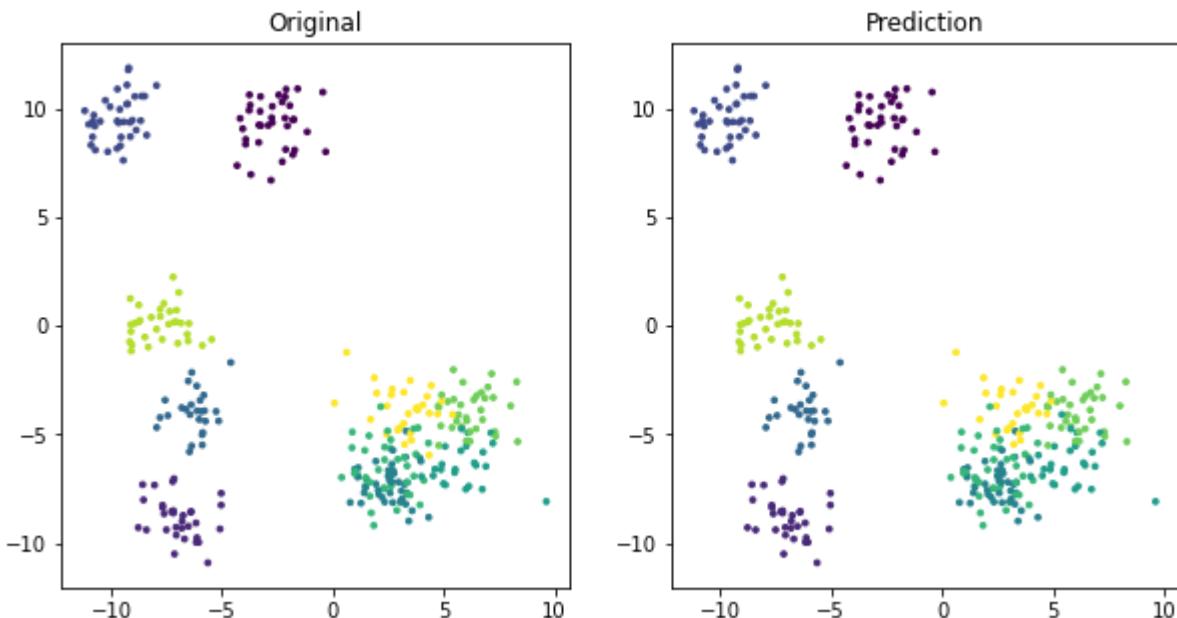
```
In [3]: from knn import KNN
knn = KNN(k=10)
knn.fit(X_train,y_train)
```

```
In [4]: y_pred = knn.predict(X_test)
```

```
In [5]: fig, axs = plt.subplots(1,2,figsize=(10,5))

axs[0].scatter(X_test[:,0],X_test[:,1],s=7,c=y_test)
axs[0].set_title('Original')
axs[1].scatter(X_test[:,0],X_test[:,1],s=7,c=y_pred)
axs[1].set_title('Prediction')
```

```
Out[5]: Text(0.5, 1.0, 'Prediction')
```



d) Wenden Sie ihren Algorithmus auf das Neutrino Monte-Carlo von Blatt 5 an.

```
In [6]: import numpy as np
import pandas as pd
```

Daten preparieren

```
In [7]: hdf_store = pd.HDFStore('NeutrinoMC.hdf5')
hdf_store.keys()
```

```
Out[7]: ['/Background', '/Signal']
```

Definitionen:

	Klasse	Label	Abkürzung
	Background	0	bg
	Signal	1	sg

In [8]:

```
df_bg = pd.read_hdf(hdf_store, key='Background')
df_bg
```

Out[8]:

	NumberOfHits	x	y
0	34.0	5.954540	4.150890
1	60.0	5.301269	4.981201
2	66.0	7.959579	2.582597
3	190.0	5.945263	9.461087
4	27.0	3.287903	5.043371
...
9999995	8.0	7.669263	3.490592
9999996	1068.0	0.726955	4.363312
9999997	7776.0	0.304958	0.646405
9999998	40.0	9.099021	5.217930
9999999	21164.0	4.834747	4.305504

10000000 rows × 3 columns

In [9]:

```
df_sg = pd.read_hdf(hdf_store, key='Signal')
df_sg
```

Out[9]:

	Energy	AcceptanceMask	NumberOfHits	x	y
0	1.564945	False	NaN	NaN	NaN
1	3.704830	False	NaN	NaN	NaN
2	1.794735	False	NaN	NaN	NaN
3	1.470759	False	NaN	NaN	NaN
4	3.930271	False	NaN	NaN	NaN
...
99995	1.080202	False	NaN	NaN	NaN
99996	1.250218	False	NaN	NaN	NaN
99997	5.154960	True	47.0	6.829988	2.523224
99998	1.012822	False	NaN	NaN	NaN

	Energy	AcceptanceMask	NumberOfHits	x	y
99999	3.043450	False	NaN	NaN	NaN

```
In [10]: hdf_store.close()
```

```
In [11]: feature_names = ['NumberOfHits', 'x', 'y']
```

```
In [12]: X_bg = df_bg[feature_names].to_numpy()
y_bg = np.zeros(shape=X_bg.shape[0])
X_bg.shape, y_bg.shape
```

```
Out[12]: ((10000000, 3), (10000000,))
```

```
In [13]: X_sg = df_sg.loc[df_sg['AcceptanceMask'], feature_names].to_numpy()
y_sg = np.ones(shape=X_sg.shape[0])
X_sg.shape, y_sg.shape
```

```
Out[13]: ((25007, 3), (25007,))
```

```
In [14]: n_train_bg = 5000
n_train_sg = 5000
n_test_bg = 20000
n_test_sg = 10000
n_bg = n_train_bg + n_test_bg
n_sg = n_train_sg + n_test_sg
```

```
In [15]: # nimm die ersten Einträge als Trainings Datensatz
X_train = np.concatenate((X_bg[:n_train_bg], X_sg[:n_train_sg]))
y_train = np.concatenate((y_bg[:n_train_bg], y_sg[:n_train_sg]))
X_train.shape, y_train.shape
```

```
Out[15]: ((10000, 3), (10000,))
```

```
In [16]: # nimm die Einträge danach als Test Datensatz
X_test = np.concatenate((X_bg[n_train_bg:n_bg], X_sg[n_train_sg:n_sg]))
y_test = np.concatenate((y_bg[n_train_bg:n_bg], y_sg[n_train_sg:n_sg]))
X_test.shape, y_test.shape
```

```
Out[16]: ((30000, 3), (30000,))
```

k-NN trainieren und testen

```
In [17]: from knn import KNN
knn = KNN(k=10)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
```

Reinheit, Effizienz und Signifikanz auf dem Test Datensatz

```
In [18]: def true_false_positiv_negative(y_true,y_pred):
    # positiv = Signal = 1
    # negativ = Background = 0
    tp = ((y_true==1) & (y_pred==1)).sum()
    fp = ((y_true==0) & (y_pred==1)).sum()
    tn = ((y_true==0) & (y_pred==0)).sum()
    fn = ((y_true==1) & (y_pred==0)).sum()
    return tp,fp,tn,fn
```

```
In [19]: def precision(tp,fp,tn,fn):
    'Reinheit'
    return tp/(tp+fp)

def recall(tp,fp,tn,fn):
    'Effizienz'
    return tp/(tp+fn)

def significance(tp,fp,tn,fn):
    'Signifikanz'
    # Wie auf Blatt06
    return tp/np.sqrt(tp+fp)
```

```
In [20]: tp,fp,tn,fn = true_false_positiv_negative(y_test,y_pred)
print(f'Reinheit: \t {precision(tp,fp,tn,fn)}')
print(f'Effizienz: \t {recall(tp,fp,tn,fn)}')
print(f'Signifikanz: \t {significance(tp,fp,tn,fn)}')
```

Reinheit: 0.8473721465227393
 Effizienz: 0.9577
 Signifikanz: 90.084865805796

e) Was ändert sich wenn Sie $\log_{10}(\text{AnzahlHits})$ statt AnzahlHits

```
In [21]: X_train_log = X_train.copy()
X_train_log[:,0] = np.log10(X_train_log[:,0])

X_test_log = X_test.copy()
X_test_log[:,0] = np.log10(X_test_log[:,0])
```

```
In [22]: knn_log = KNN(k=10)
knn_log.fit(X_train_log,y_train)
y_pred_log = knn_log.predict(X_test_log)
```

In [23]:

```
tp,fp,tn,fn = true_false_positiv_negative(y_test,y_pred_log)
print(f'Reinheit: \t {precision(tp,fp,tn,fn)}')
print(f'Effizienz: \t {recall(tp,fp,tn,fn)}')
print(f'Signifikanz: \t {significance(tp,fp,tn,fn)}')
```

Reinheit: 0.8752237737200144
 Effizienz: 0.9778
 Signifikanz: 92.50912419558571

f) Was ändert sich, wenn Sie $k = 20$ statt $k = 10$ verwenden?

In [24]:

```
knn_20 = KNN(k=20)
knn_20.fit(X_train,y_train)
y_pred_20 = knn_20.predict(X_test)
```

In [25]:

```
tp,fp,tn,fn = true_false_positiv_negative(y_test,y_pred_20)
print(f'Reinheit: \t {precision(tp,fp,tn,fn)}')
print(f'Effizienz: \t {recall(tp,fp,tn,fn)}')
print(f'Signifikanz: \t {significance(tp,fp,tn,fn)}')
```

Reinheit: 0.8235597592433362
 Effizienz: 0.9578
 Signifikanz: 88.81472498427654

$k = 20$ und $\log_{10}(\text{AnzahlHits})$

In [26]:

```
knn_20 = KNN(k=20)
knn_20.fit(X_train_log,y_train)
y_pred_20_log = knn_20.predict(X_test_log)
```

In [27]:

```
tp,fp,tn,fn = true_false_positiv_negative(y_test,y_pred_20_log)
print(f'Reinheit: \t {precision(tp,fp,tn,fn)}')
print(f'Effizienz: \t {recall(tp,fp,tn,fn)}')
print(f'Signifikanz: \t {significance(tp,fp,tn,fn)}')
```

Reinheit: 0.8629704360031581
 Effizienz: 0.9837
 Signifikanz: 92.13598742599477

Übersicht:

k	log10?	Reinheit	Effizienz	Signifikanz
10	Nein	0.847	0.9577	90.1
10	Ja	0.875	0.9778	92.5
20	Nein	0.824	0.9578	88.8

k	log10?	Reinheit	Effizienz	Signifikanz
---	--------	----------	-----------	-------------

20	Ja	0.863	0.9837	92.1
----	----	-------	--------	------

Wenn man `log10(AnzahlHits)` statt `AnzahlHits` verwendet steigen alle Metriken, da hier das in a) angesprochene Problem gelöst wird.

Wenn man $k = 20$ statt $k = 10$ verwendet sinken Reinheit und Signifikanz, aber die Effizienz

In []:

./smd/blatt10/knn.py
2021-07-06T14:08+02:00

```
1 import numpy as np
2 from tqdm.notebook import tqdm
3
4 class KNN:
5     '''KNN Classifier.
6
7     Attributes
8     -----
9     k : int
10        Number of neighbors to consider.
11    '''
12    def __init__(self, k):
13        '''Initialization.
14        Parameters are stored as member variables/attributes.
15
16        Parameters
17        -----
18        k : int
19            Number of neighbors to consider.
20        '''
21        self.k = k
22
23    def fit(self, X, y):
24        '''Fit routine.
25        Training data is stored within object.
26
27        Parameters
28        -----
29        X : numpy.array, shape=(n_samples, n_attributes)
30            Training data.
31        y : numpy.array shape=(n_samples)
32            Training labels.
33        '''
34        self.X_ = X
35        self.y_ = y
36
37    def predict(self, X):
38        '''Prediction routine.
39        Predict class association of each sample of X.
40
```

```

41      Parameters
42      -----
43      X : numpy.array, shape=(n_samples, n_attributes)
44          Data to classify.
45
46      Returns
47      -----
48      prediction : numpy.array, shape=(n_samples)
49          Predictions, containing the predicted label of each sample.
50          ''
51      prediction = np.empty(shape=(X.shape[0]))
52      for i,x in enumerate(tqdm(X)):
53          prediction[i] = self.predict_single(x)
54
55      return prediction
56
57  def predict_single(self,x):
58      '''Predict routine for a single sample
59      Predict class association for a single sample x
60
61      Parameters
62      -----
63      x : numpy.array, shape=(n_attributes,)
64          Data to classify.
65
66      Returns
67      -----
68      prediction : int
69          Prediction, containing the predicted label of the sample.
70          ''
71      x = x.reshape(1,-1)
72      # Calculate all distances to every sample in the training
73      # dataset
74      distances = np.linalg.norm(self.X_ - x, axis=1)
75
76      # find the k nearest neighbors
77      k_nearest = np.argsort(distances)[:self.k]
78
79      # associate the best fitting class label
80      y_unique, y_counts = np.unique(self.y_[k_nearest],
81          return_counts=True)
82      return y_unique[np.argmax(y_counts)]

```

Aufgabe 21: k-Means per Hand

Loyds - Algorithmus:

1) Wähle k Start - Clusterzentren $\vec{c}_n^{(0)}$

2) Ordne die Daten nach Abstand $|\vec{c}_n^{(i)} - \vec{x}_m|$
zu den k Clustern zu

3) Berechne die neuen Clusterzentren
als den Schwerpunkt des Clusters

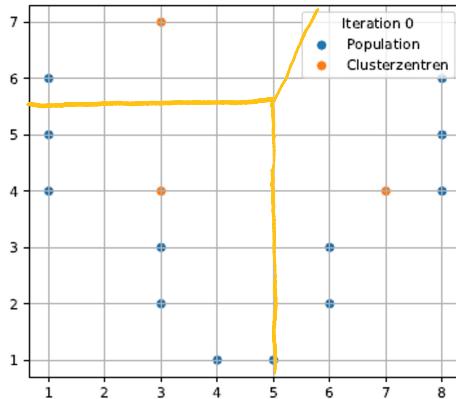
$$\vec{c}_n^{(i+1)} = \frac{1}{N_n} \sum_{m=0}^{N_n} \vec{x}_m$$

4) nächste Iteration startet bei 2)

Population: (1;4) (1;5) (1;6) (3;3) (3;2) (4;1) (5;1) (6;2) (6;3) (8;4) (8;5) (8;6)

a)

1)



Zuordnung zu Clustern: (bei (5,1) ist der Abstand gleich)

$$\vec{c}_1 = (3, 7) : (1, 6)$$

$$\vec{c}_1 = (3, 7) : (1, 6)$$

$$\vec{c}_2 = (3, 4) : (1, 5); (1, 4); (3, 3); (3, 2); (4, 1); (5, 1)$$

$$\vec{c}_3 = (7, 4) : (6, 2); (6, 3); (8, 4); (8, 5); (8, 6)$$

neue Clusterzentren:

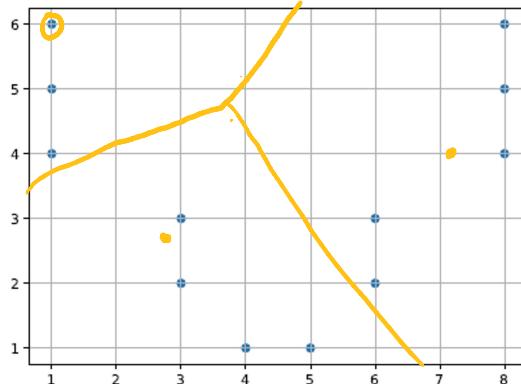
$$\vec{c}_1 = (1, 6)$$

$$\vec{c}_2 = \left(\frac{17}{6}, \frac{8}{3} \right) = (2.8\bar{3}, 2.\bar{6})$$

$$\vec{c}_3 = \left(\frac{36}{5}, 4 \right) = (7.2, 4)$$

b)

2)



Zuordnung zu Clustern:

$$\vec{c}_1 = (1, 6) : (1, 6); (1, 5); (1, 4)$$

$$\vec{c}_2 = (2.8\bar{3}, 2.\bar{6}) : (3, 3); (3, 2); (4, 1); (5, 1)$$

$$\vec{c}_3 = (7.2, 4) : (6, 2); (6, 3); (8, 4); (8, 5); (8, 6)$$

Abstände die nicht klar abzulesen sind:

$$|(1,4) - \vec{c}_1| \approx 2$$

$$|(1,4) - \vec{c}_1| = 2$$

$$|(1,4) - \vec{c}_2| = 3,8$$

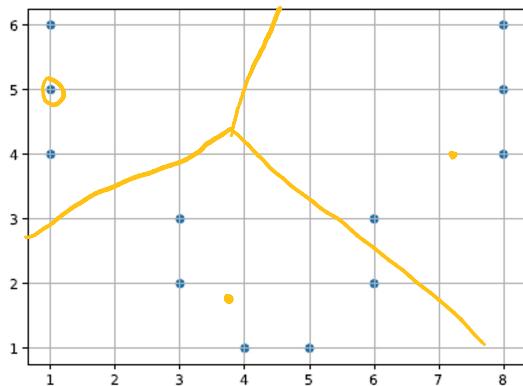
neue Clusterzentren:

$$\vec{c}_1 = (1, 5)$$

$$\vec{c}_2 = \left(\frac{15}{4}, \frac{7}{4}\right) = (3.75, 1.75)$$

$$\vec{c}_3 = \left(\frac{26}{5}, 4\right) = (7.2, 4)$$

3)



Zuordnung zu Clustern:

$$\vec{c}_1 = (1, 5) : (1, 6); (1, 5); (1, 4)$$

$$\vec{c}_2 = (3.75, 1.75) : (3, 3); (3, 2); (4, 1); (5, 1); (6, 2)$$

$$\vec{c}_3 = (7.2, 4) : (6, 3); (8, 4); (8, 5); (8, 6)$$

Abstände die nicht klar abzulesen sind:

$$|(6,2) - \vec{c}_2| = 2.264$$

$$|(6,2) - \vec{c}_3| = 2.332$$

neue Clusterzentren:

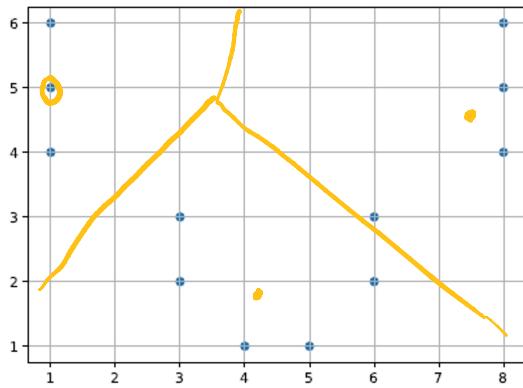
neue Clusterzentren:

$$\vec{c}_1 = (1, 5)$$

$$\vec{c}_2 = \left(\frac{21}{5}, \frac{9}{5}\right) = (4.2, 1.8)$$

$$\vec{c}_3 = \left(\frac{15}{2}, \frac{9}{2}\right) = (7.5, 4.5)$$

4)



Zuordnung zu Clustern:

$$\vec{c}_1 = (1, 5) : (1, 4); (1, 5); (1, 6)$$

$$\vec{c}_2 = (4.2, 1.8) : (3, 3); (3, 2); (4, 1); (5, 1); (6, 2)$$

$$\vec{c}_3 = (7.5, 4.5) : (6, 3); (8, 4); (8, 5); (8, 6)$$

Abstände die nicht klar abzuksen sind:

$$|(6, 3) - \vec{c}_2| = 2.163$$

$$|(6, 3) - \vec{c}_3| = 2.121$$

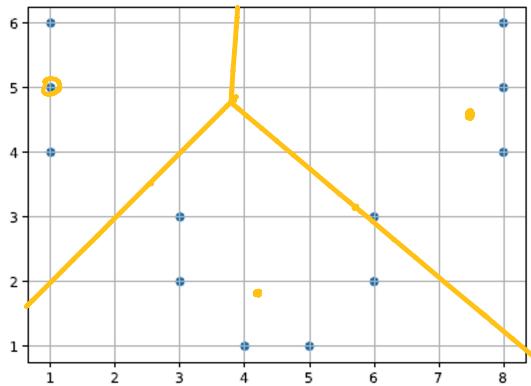
An den Clusterzuordnungen hat sich nichts geändert.

Dadurch bleiben auch die Clusterzentren gleich.

Also ist der Algorithmus in ein Gleichgewicht konvergiert.

Finale Zuordnung:

$$\vec{c}_1 = (1, 5); \vec{c}_2 = (4.2, 1.8); \vec{c}_3 = (7.5, 4.5)$$



c)

Nach 3 bzw. 4 Iterationen ist der Algorithmus konvergiert.

Das Ergebnis entspricht nicht unseren Erwartungen,
denn den Punkt (6, 3) hätten wir zum mittleren
Cluster zugeordnet.

Man sieht, dass k-Means teilweise stark
von der Wahl der Initial-Clusterzentren abhängt.

Unsere Einschätzung / Erwartung:

$$\vec{c}_1 = (1, 5); \vec{c}_2 = (4.5, 2); \vec{c}_3 = (8, 5)$$

