TSRI-7

July 23, 2024

1 Übung 7

Gruppenname: TSRI

- Christian Rene Thelen @cortex359
- Leonard Schiel @leo_paticumbum
- Marine Raimbault @Marine Raimbault
- Alexander Ivanets @sandrium

1.0.1 In dieser Übung ...

... werden Sie zunächst Multidimensionales Skalieren (MDS) implementieren und anwenden (Übung 7.1). MDS wird dann der Ausgangspunkt sein, dass Sie eine nichtlineare Dimensionsreduktion mit Isomap (Übung 7.2) durchführen können.

1.0.2 7.1 Städte in Deutschland (Multidimensionales Skalieren)

In dieser Übung werden Sie Multidimensionales Skalieren implementieren und damit die Daten einer Distanzmatrix visualisieren: Sie erhalten die Entfernung (in Autobahnkilometer) zwischen deutschen Großstädten und finden mithilfe von MDS Koordinaten, mit denen Sie die Position dieser Städte visualisieren können.

Ihre Aufgaben

Bearbeiten Sie bitte alle Aufgaben mit *Numpy* und die Visualisierung später mit *Matplotlib*.

(1) Importieren Sie Ihre Daten, indem Sie die unten stehende Code-Zelle ausführen.

[1]:
```
import numpy as np
from matplotlib import pyplot as plt

# Matrix der Distanzen (in Autobahnkilometern)
M: np.ndarray = np.array([[0, 548, 289, 576, 586],
              [548, 0, 493, 195, 392],
              [289, 493, 0, 427, 776],
              [576, 195, 427, 0, 577],
              [586, 392, 776, 577, 0]])

# Zugehörige Labels
labels = ['Berlin', 'Frankfurt', 'Hamburg', 'Köln', 'München']
```

(2) Schlagen Sie in den Vorlesungsfolien nach, wie die Multidimensionale Skalierung in einzelnen Schritten durchgeführt wird.

Die Matrix `M` enthält Distanzen. Aber die Multidimensionale Skalierung erwartet eine Distanzmatrix `D`, in der die Distanzen quadriert vorliegen. Erzeugen Sie nun die Distanzmatrix `D`.

[2]:
```
D: np.ndarray = np.multiply(M, M)
D
```

[2]:
```
array([[     0, 300304,  83521, 331776, 343396],
       [300304,      0, 243049,  38025, 153664],
       [ 83521, 243049,      0, 182329, 602176],
       [331776,  38025, 182329,      0, 332929],
       [343396, 153664, 602176, 332929,      0]])
```

(3) Erzeugen Sie die Matrix `B` mithilfe der Matrix `D`. Dieser Schritt wird auch *Double Centering* (Doppelzentrierung) genannt.

[3]:
```
N: int = D.shape[0]
H: np.ndarray = np.identity(N) - 1/N * np.ones((N, N))
B: np.ndarray = -0.5 * H @ D @ H
B
```

[3]:
```
array([[ 107352.64,  -75194.86,   70799.94,  -75929.16,  -27028.56],
       [ -75194.86,   42561.64,  -41359.56,   38550.84,   35441.94],
       [  70799.94,  -41359.56,  117768.24,    4002.14, -151210.76],
       [ -75929.16,   38550.84,    4002.14,   72565.04,  -39188.86],
       [ -27028.56,   35441.94, -151210.76,  -39188.86,  181986.24]])
```

(4) Führen Sie nun eine Eigenwertzerlegung der Matrix `B` durch.

- Sortieren Sie die Eigenwerte in absteigender Reihenfolge.
- Sortieren Sie die Eigenvektoren in der Reihenfolge der Eigenwerte. Beachten Sie, dass die Eigenvektoren *den Spalten der von Numpy zurückgegebenen Matrix* entsprechen.

[4]:
```
eig_vals, eig_vecs = np.linalg.eigh(B)
idx = np.argsort(eig_vals)[::-1]
eig_vals, eig_vecs = eig_vals[idx], eig_vecs[:, idx]
```

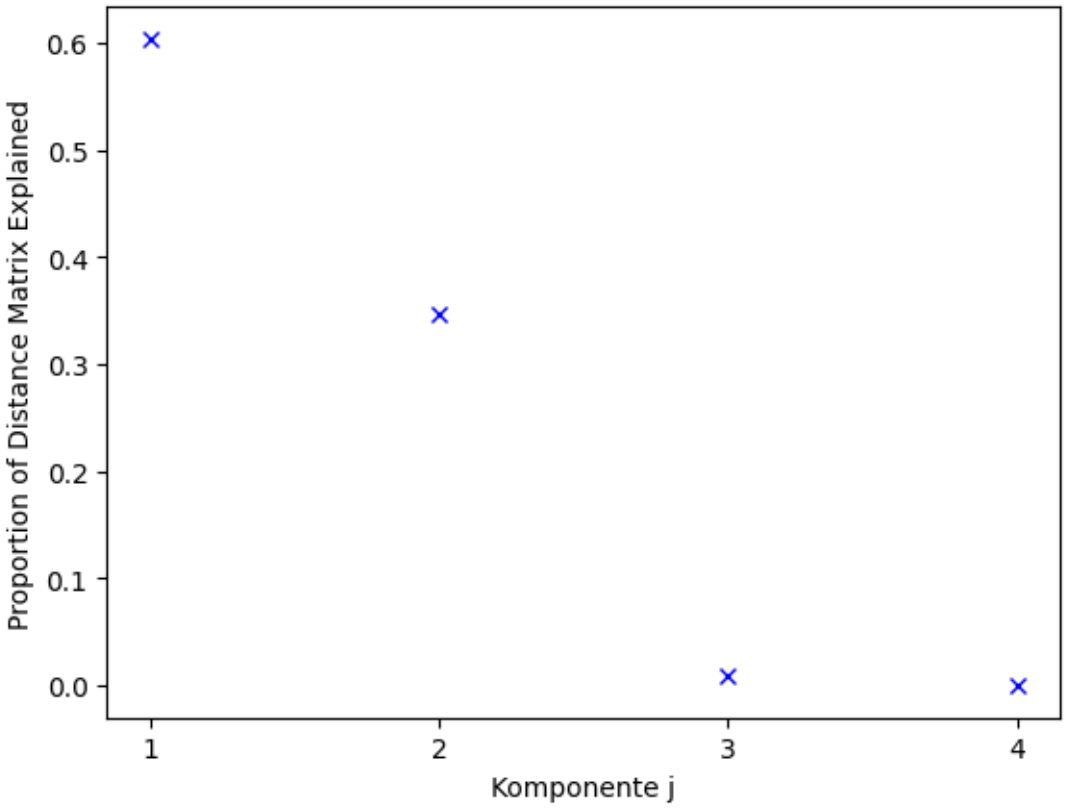
(5) Berechnen Sie die *Proportion of Distance Matrix Explained* (PDME).

- Können Sie einen Ellbogenverlauf erkennen?
- Wie viele Dimensionen (Komponenten) sollte jeder Datenpunkt haben, dessen Koordinaten Sie rekonstruieren werden?

Proportion of Distance Matrix Explained

$$\text{PDME}(j) = \frac{\lambda_j}{\sum_{i=1}^{N} |\lambda_i|} \quad \forall j \text{ mit } \lambda_j \geq 0$$

[5]:
```
pdme = eig_vals / np.abs(eig_vals).sum()
pdme = pdme[pdme >= 0]
plt.plot(np.arange(len(pdme)) + 1, pdme, "bx")
plt.xticks(np.arange(len(pdme)) + 1)
plt.xlabel("Komponente j")
plt.ylabel("Proportion of Distance Matrix Explained")
plt.show()
```



Es lässt sich ein Ellbogenverlauf erkennen und für die Anzahl der Dimensionen lässt sich klar ablesen, dass diese 2 betragen sollte.

(6) Eliminierung negativer Eigenwerte:

- Haben Sie in Schritt 4 negative Eigenwerte beobachtet? Verwerfen Sie diese mitsamt den assoziierten Eigenvektoren. Am Ende der Übung erkläre ich Ihnen, warum Sie negative Eigenwerte beobachten könnten.

[6]:
```
# Reiehnfolge beachten!
eig_vecs = eig_vecs[:, eig_vals >= 0]
eig_vals = eig_vals[eig_vals >= 0]
```

(7) Konstruieren Sie mit den Eigenvektoren und Eigenwerten aus Schritt (6) die Datenmatrix `X`. Schlagen Sie dazu in der Vorlesung nach, wie dies geht. Wählen Sie für die Konstruktion so

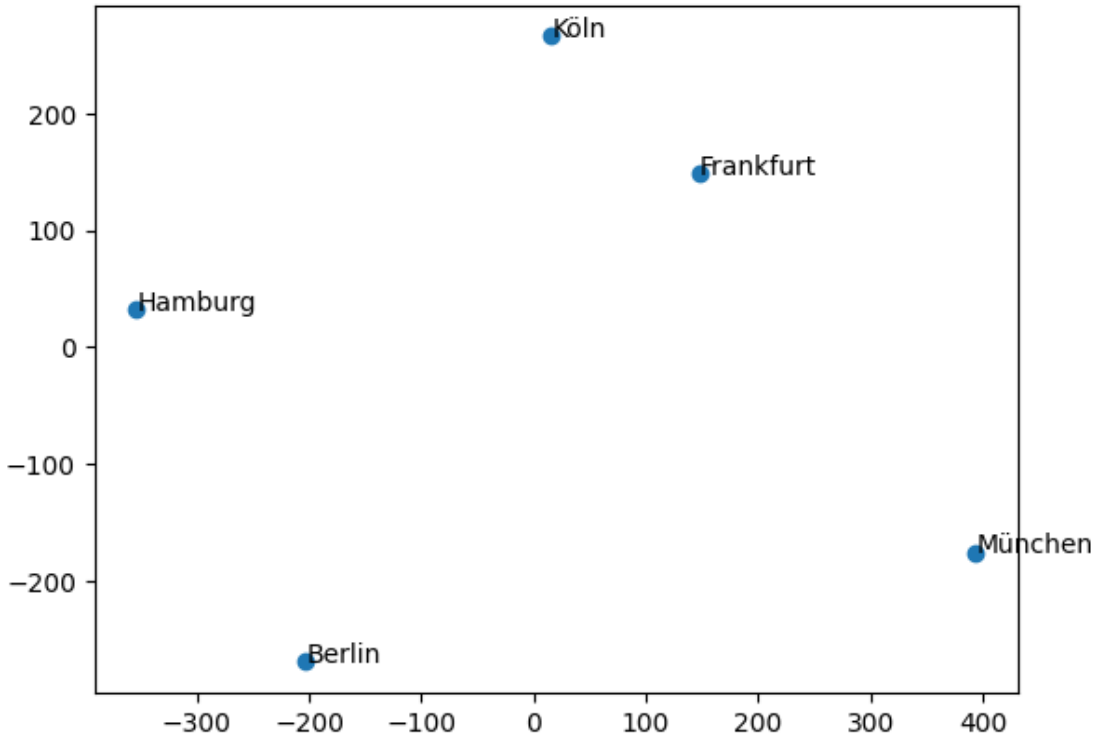
viele Features (Dimensionen) aus, wie Sie in Schritt (5) über die PDME ermittelt haben.

[7]:
```
Q: int = 2 # Anzahl der Dimensionen nach Dimensionsreduktion
X: np.ndarray = np.diag(np.sqrt(eig_vals[:Q])) @ eig_vecs[:, :Q].T
X
```

[7]:
```
array([[-203.18600393,  147.7359528 , -353.99640444,   15.78611067,
         393.66034489],
       [-269.50533465,  148.06345119,   32.03109021,  265.94192159,
        -176.53112834]])
```

(8) Visualisieren Sie die Daten in den von Ihnen in Schritt (7) ermittelten Koordinaten. Beschriften Sie jeden Datenpunkt mit dem Namen der jeweiligen Stadt, die er repräsentiert.

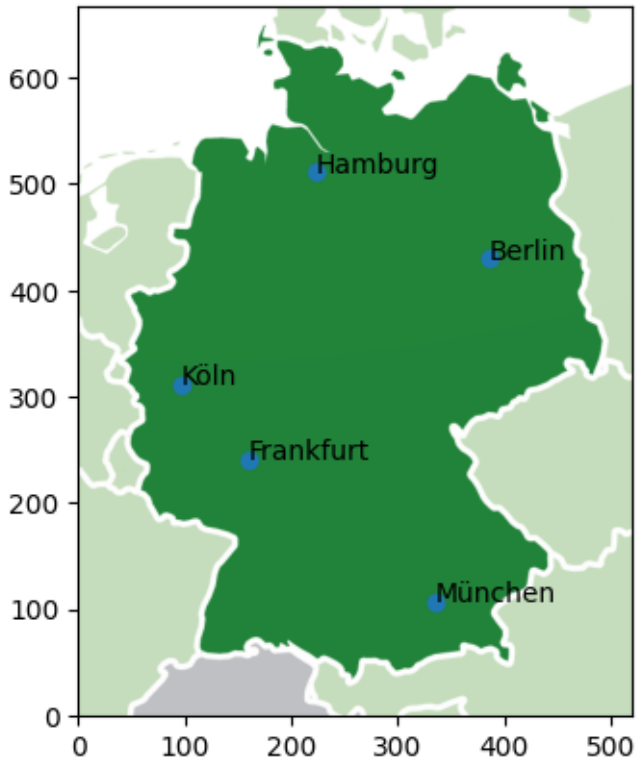
[8]:
```
plt.scatter(*X)
for i in range(X.shape[1]):
    plt.annotate(labels[i], X[:, i])
plt.show()
```



(9) Ähnelt Ihre Abbildung aus Schritt (8) der geografischen Lage dieser deutschen Städte? Falls Sie Unterschiede bemerken, woran könnten diese Unterschiede liegen? (1-3 Sätze)

Die Abbildung zeigt die Abstände und relative Lage der Städte zueinander (Distanzen invariant unter orthogonalen Transformationen (Rotation, Spiegelung) sowie Koordinatenursprungsverschiebung).

[9]:
```
im = plt.imread("germany.png", ) # Quelle: https://upload.wikimedia.org/
  ↪wikipedia/commons/2/26/EU-Germany.svg
plt.imshow(np.flipud(im), origin="lower")

phi = 1.5 * np.pi
rotation = np.array([
    [np.cos(phi), -np.sin(phi)],
    [np.sin(phi), np.cos(phi)]
])
spiegelung = np.array([
    [-1, 0],
    [0, 1]
])

X_adjusted = X.T * 0.54
X_rotated = (X_adjusted @ spiegelung @ rotation + np.array([[240, 320]])).T

plt.scatter(*X_rotated)
for i in range(X_rotated.shape[1]):
    plt.annotate(labels[i], X_rotated[:, i])
plt.show()
```



Anmerkung zum Auftreten negativer Eigenwerte

Wenn die Matrix B aus euklidischen Distanzen hergeleitet wurde, gilt $B = XX^{\mathrm{T}}$. In diesem Fall ist die Matrix B positiv-semidefinit:

$$\mathbf{a}^{\mathrm{T}} B\mathbf{a} = \mathbf{a}^{\mathrm{T}} XX^{\mathrm{T}}\mathbf{a} = (X\mathbf{a})^{\mathrm{T}} (X\mathbf{a}) \geq 0 \qquad \forall \mathbf{a}$$

Damit gilt insbesondere für alle ihre Eigenwerte: $\lambda_i \geq 0$.

Wenn Sie nun für eine Matrix B, die Sie aus einer Matrix D konstruiert haben, negative Eigenwerte beobachten, bedeutet dies, dass die Matrix D keine "euklidische Distanzmatrix" ist, d.h. keine euklidischen Distanzen zwischen Objekten enthält. Dies ist nicht weiter schlimm. In der Praxis werden negative Eigenwerte und assoziierte Eigenvektoren verworfen. Daneben wird für die Berechnung der PDME im Nenner die Absolutbeträge der Eigenwerte verwendet.

1.0.3 7.2 Schweizer Rollkuchen (nichtlineare Dimensionsreduktion mit Isomap)

Die PCA erlaubt Ihnen die Extraktion von linearen Strukturen aus hochdimensionalen Räumen. Allerdings liegen in der Realität Daten auf nichtlinearen Mannigfaltigkeiten vor. In dieser Übung lernen Sie Isomap kennen, mithilfe dessen Sie genau solche nichtlineare Strukturen extrahieren können. Das Paper zu Isomap wurde seinerzeit im renommierten Magazin *Science* publiziert. Isomap ist eine Variante des Multidimensionalen Skalierens, allerdings werden die paarweisen Distanzen zwischen den Datenpunkten auf eine andere Art bestimmt als in der klassischen MDS.

Ihre Daten

Sie bearbeiten einen klassischen Datensatz: Den sogenannten *Schweizer Rollkuchen* (swiss roll). Wenn Sie wissen wollen, wie dieser Kuchen aussieht, schauen Sie doch zum Beispiel hier vorbei.

Ihre Aufgaben

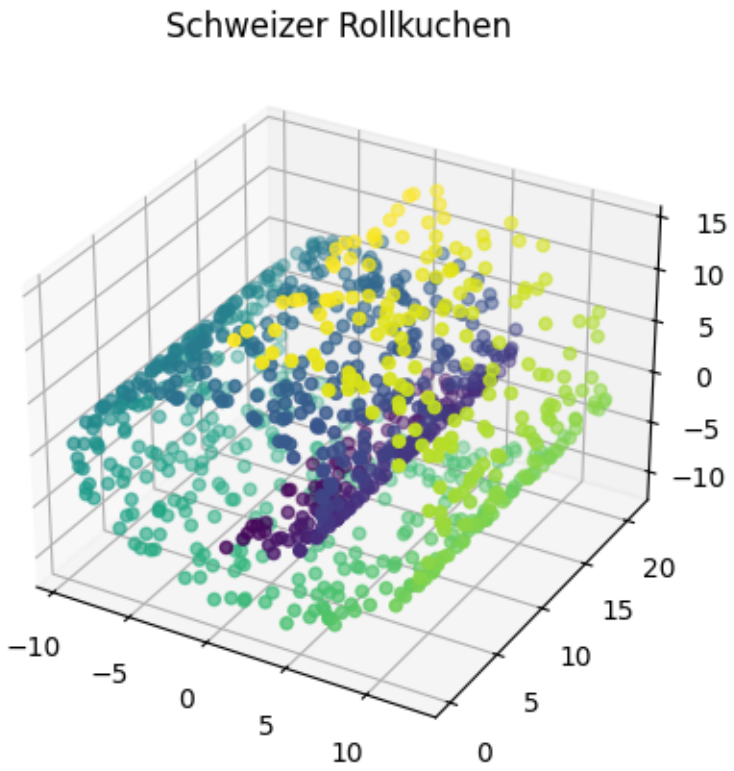
- Nutzen Sie im Nachfolgenden Ihren Code aus Übung 7.1 (MDS), um Isomap zu implementieren.

(1) Führen Sie die unten stehende Code-Zelle aus, um den Datensatz zu erzeugen und um Bibliotheken zu importieren.

[10]:
```
import numpy as np
from matplotlib import pyplot as plt
from sklearn.datasets import make_swiss_roll
from sklearn.metrics import pairwise_distances
from scipy.sparse.csgraph import shortest_path
from sklearn.neighbors import kneighbors_graph
from mpl_toolkits.mplot3d import Axes3D

# Ihre Daten
X, t = make_swiss_roll(1000, random_state=100)  # X: Koordinaten, t: Variable,␣
 ↪die die Farbe jedes Datenpunktes kodiert
```

(2) Visualisieren Sie zunächst die Daten in einem dreidimensionalen Scatterplot. Achten Sie darauf, dass Sie die Datenpunkte farbkodieren gemäß ihrer Variablen `t`.

[11]:
```
%matplotlib inline
ax = plt.axes(projection='3d')
ax.scatter3D(*X.T, c=t)

plt.title("Schweizer Rollkuchen")
plt.show()
```



(3) Schlagen Sie in der Vorlesung nach, wie Isomap in einzelnen Schritten durchgeführt wird.

- Bestimmen Sie zunächst eine Matrix $M = (m_{ij})$, wobei m_{ij} der euklidischen Distanz zwischen Datenpunkt i und j entspricht.

  Wichtig: Hier sind die Distanzen euklidisch und noch nicht quadriert.

[12]:
```
# Distanzmatirx M = (m_{ij}), wobei m{ij} die euklidische Distanz (nicht␣
↪quadriert) zwischen Datenpunkt i und j ist.
M = np.linalg.norm(X[:, np.newaxis] - X[np.newaxis, :], axis=-1)
```

(4) Erzeugen Sie aus M einen K-nächste-Nachbarn Graphen.

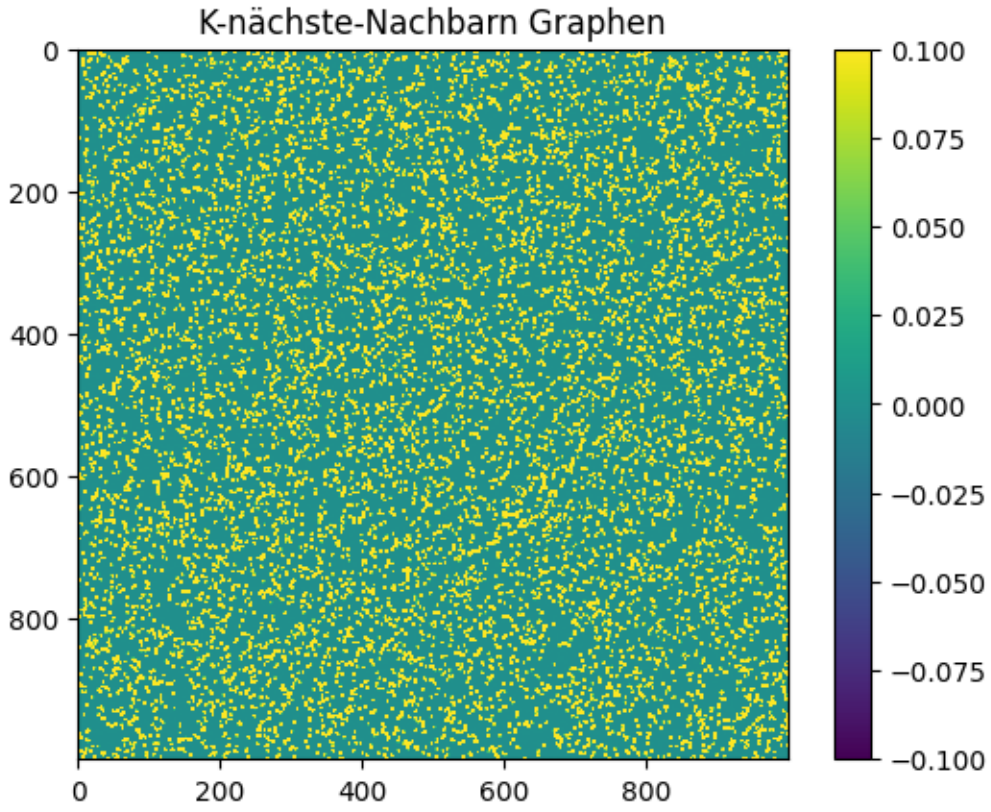
- Nutzen Sie dafür aus scikit-learn den Befehl kneighbors_graph und nutzen Sie für die Anzahl nächster Nachbarn zunächst $K = 10$.
- Achten Sie darauf, dass bei der Erstellung des Graphen die euklidischen Abstände zurückgegeben werden. Sie müssen dazu neben M und K **einen zusätzlichen Parameter** der

Funktion anpassen.

- Die entstehende Matrix ist *sparse*. Wenn Sie sich diese Matrix einmal visualisieren wollen, müssen Sie sie in eine gewöhnliche Form transformieren.

[13]:
```
from sklearn.neighbors import kneighbors_graph
knn_graph = kneighbors_graph(M, n_neighbors=10, mode='distance')
```

[14]:
```
from scipy.sparse import csr_matrix

plt.imshow(csr_matrix.todense(knn_graph), norm='linear')

plt.colorbar()
plt.title("K-nächste-Nachbarn Graphen")
plt.show()
```

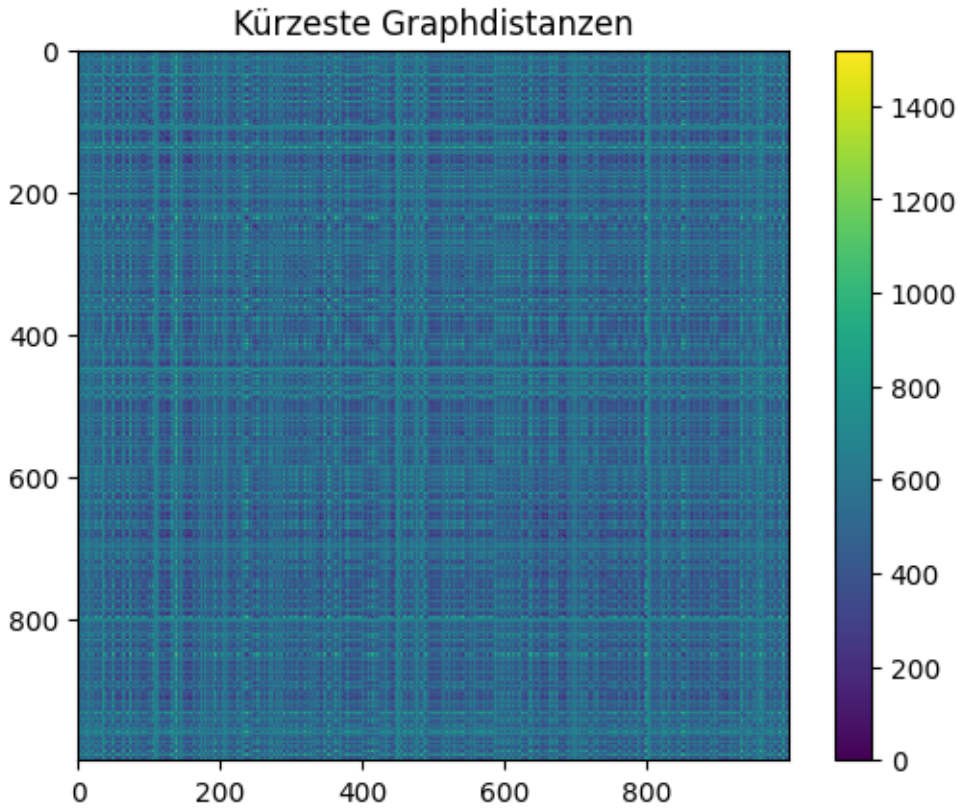


(5) Ermittelten Sie die Matrix Y der kürzesten Graphdistanzen für alle Paare von Punkten.

- Nutzen Sie dafür aus scipy den Befehl shortest_path.
- Wenn Sie wollen, können Sie sich die Matrix dieses Graphen einmal visualisieren.

[15]:
```
from scipy.sparse.csgraph import shortest_path
Y = shortest_path(knn_graph)
```

```
plt.imshow(Y, norm='linear')
plt.colorbar()
plt.title("Kürzeste Graphdistanzen")
plt.show()
```



(6) Bestimmen Sie für Y eine neue Datenmatrix (Koordinaten) Z mithilfe einer Multidimensionalen Skalierung (MDS). Nutzen Sie dafür Ihren Code aus Übung 7.1.

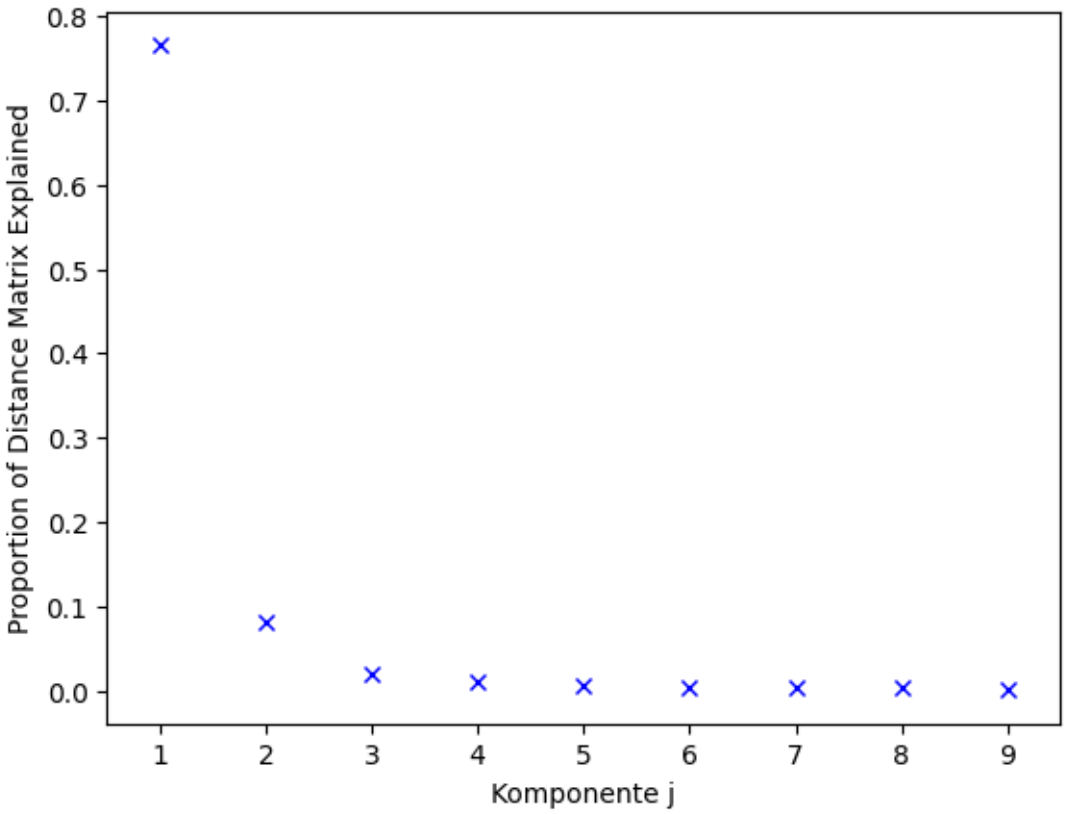
- Erinnern Sie sich: Der erste Schritt der MDS besteht darin, die Einträge einer Distanzmatrix zu quadrieren, bevor die Matrix B bestimmt wird. Vergessen Sie also nicht, die Einträge der Matrix Y zu quadrieren.

[16]:
```
def multidimensional_scaling(M: np.ndarray, Q: int) -> tuple[np.ndarray, np.
↪ndarray]:
    # Distanzmatrix quadrieren
    D: np.ndarray = np.multiply(M, M)

    # Double Centering
    N: int = D.shape[0]
    H: np.ndarray = np.identity(N) - 1/N * np.ones((N, N))
```

```
    B: np.ndarray = -0.5 * H @ D @ H

    # Eigenwertzerlegung
    eig_vals, eig_vecs = np.linalg.eigh(B)
    idx = np.argsort(eig_vals)[::-1]
    eig_vals, eig_vecs = eig_vals[idx], eig_vecs[:, idx]

    # Negative Eigenwerte eliminieren
    eig_vecs = eig_vecs[:, eig_vals >= 0]
    eig_vals = eig_vals[eig_vals >= 0]

    # Neue Datenmatrix Z aus den Eigenwerten und Eigenvektoren mit Q␣
↪Dimensionen konstruieren
    Z: np.ndarray = np.diag(np.sqrt(eig_vals[:Q])) @ eig_vecs[:, :Q].T
    return Z, eig_vals

Z, eig_vals = multidimensional_scaling(Y, 2)
```

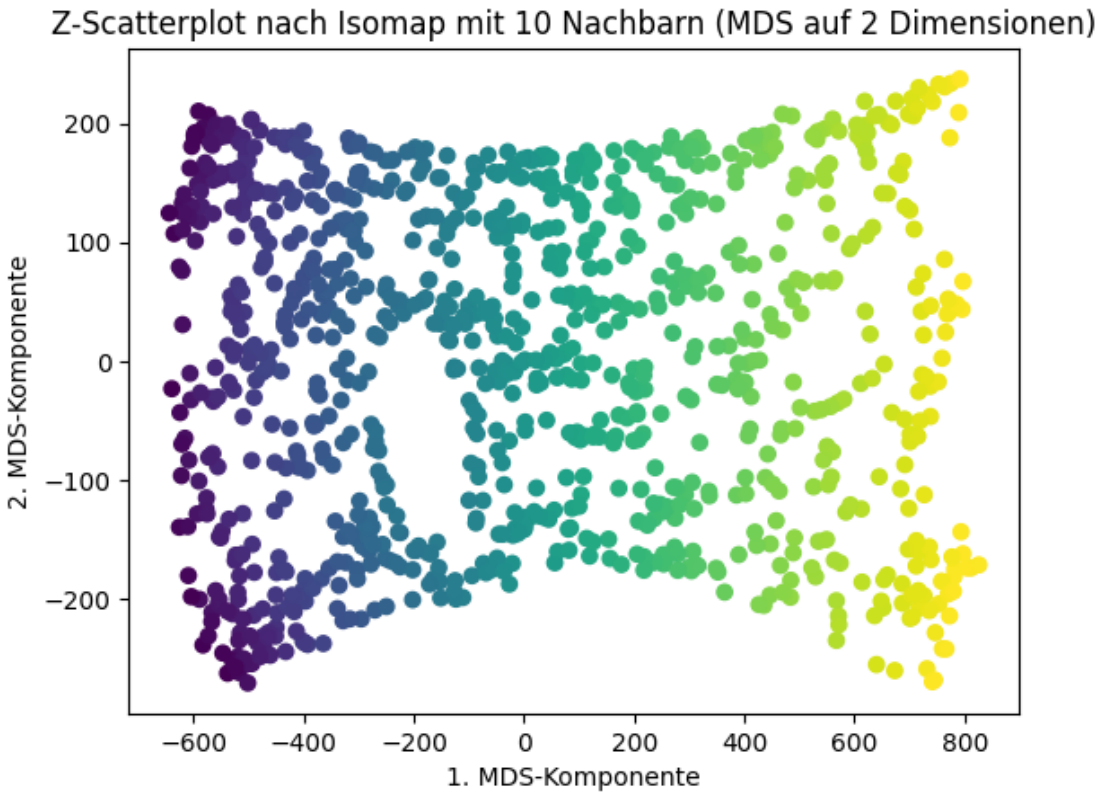
(7) Untersuchen Sie die *Proportion of Distance Matrix Explained* (PDME): Wie hochdimensional wird die nichtlineare Struktur sein, die in den Daten vorhanden ist? (1-2 Worte)

[17]:
```
pdme = eig_vals / np.abs(eig_vals).sum()
pdme = pdme[pdme >= 0]
plt.plot(np.arange(len(pdme)) + 1, pdme, "bx")
plt.xticks(np.arange(len(pdme)) + 1)
plt.xlim(0.5, 9.5)
plt.xlabel("Komponente j")
plt.ylabel("Proportion of Distance Matrix Explained")
plt.show()
```



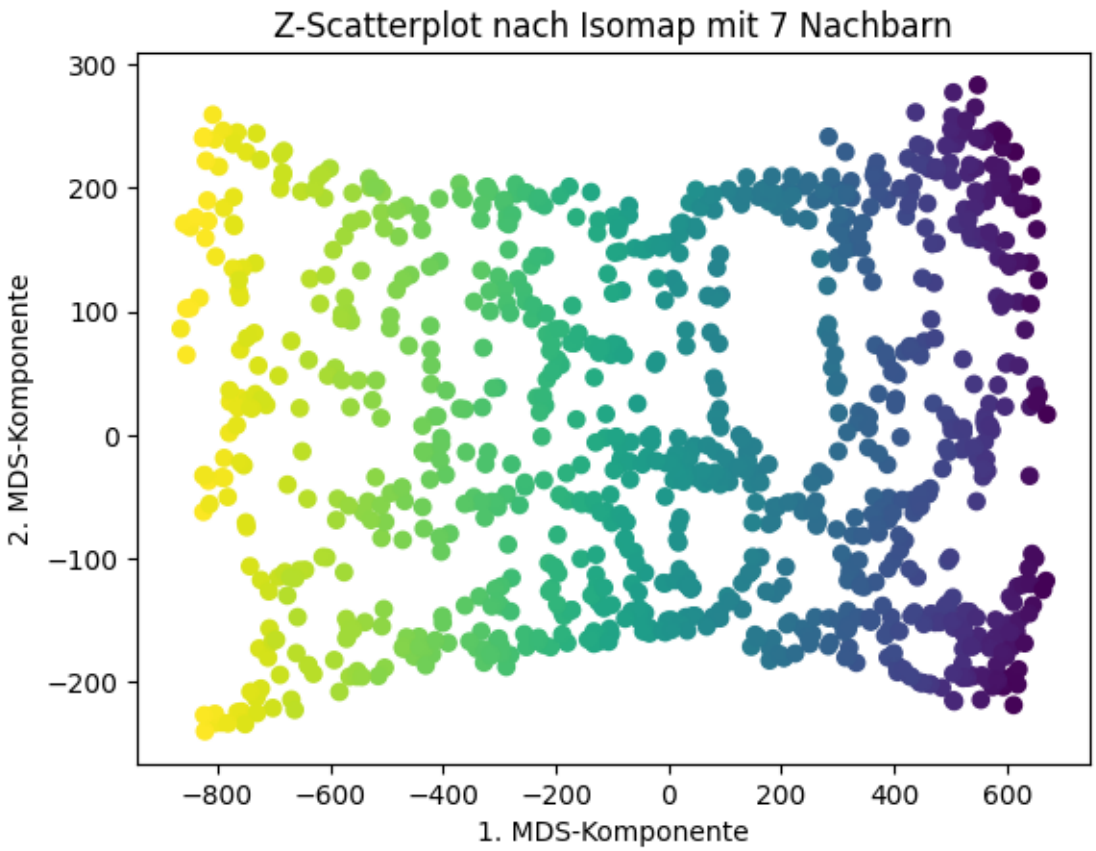
(8) Visualisieren Sie die Daten mithilfe der Koordinaten Z und farbkodieren Sie sie gemäß `t`.

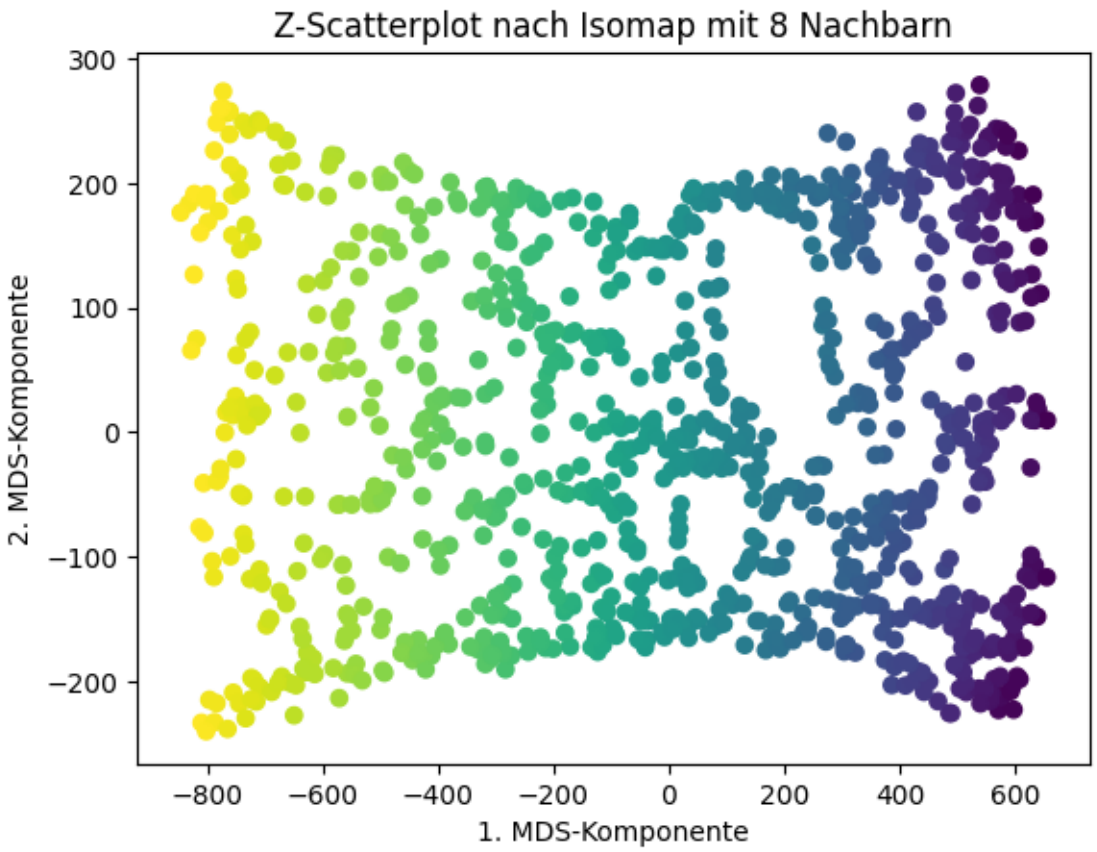
[18]:
```
plt.scatter(*Z, c=t)

plt.title(f"Z-Scatterplot nach Isomap mit 10 Nachbarn (MDS auf {Q}␣
 ↪Dimensionen)")
plt.xlabel("1. MDS-Komponente")
plt.ylabel("2. MDS-Komponente")
plt.show()
```

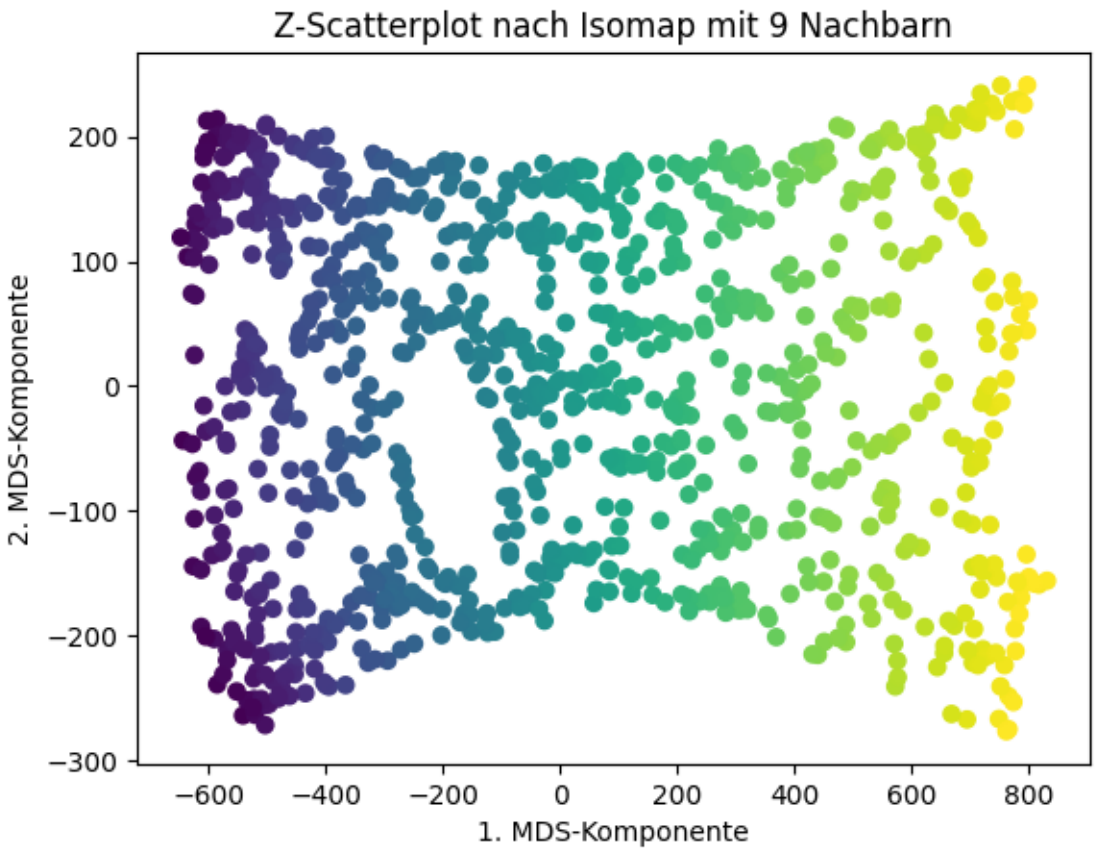


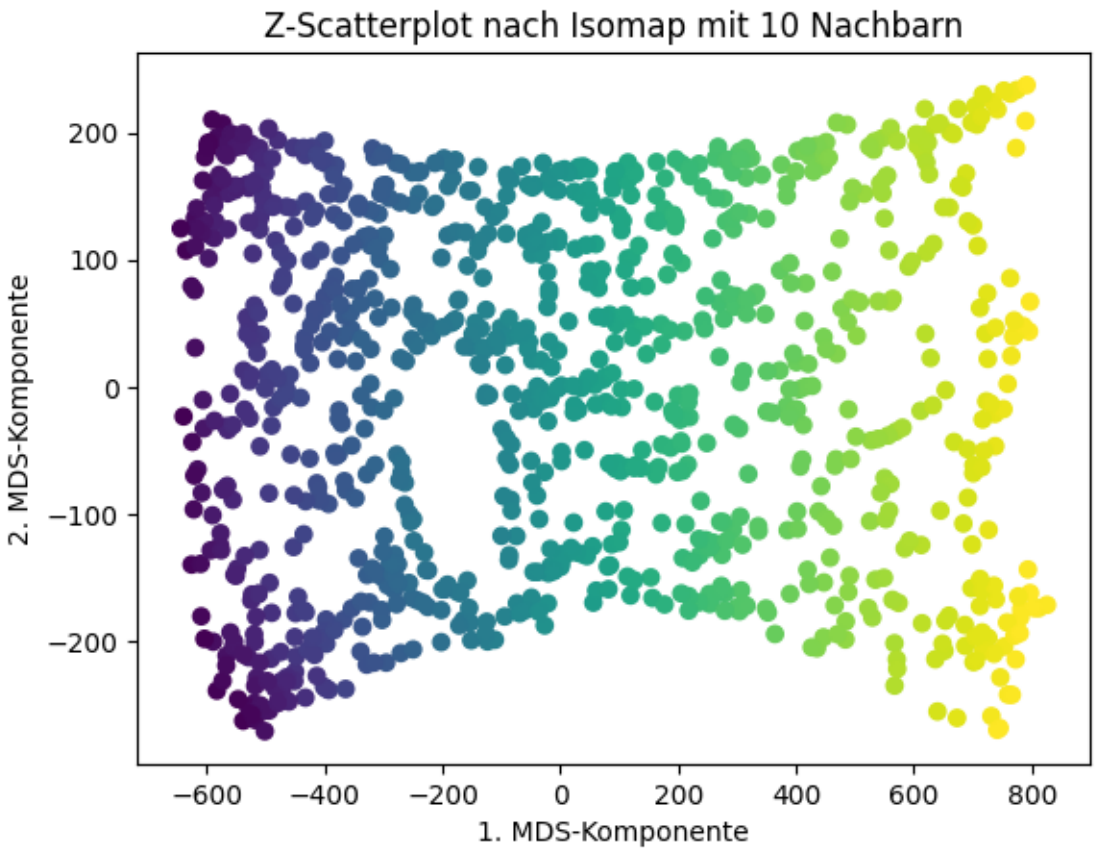
(9) [Optional] Erzeugen Sie Abbildungen wie in Schritt (8), aber für verschiedene Werte von K für die Konstruktion des Nachbarschaftsgraphen. Was fällt Ihnen auf?

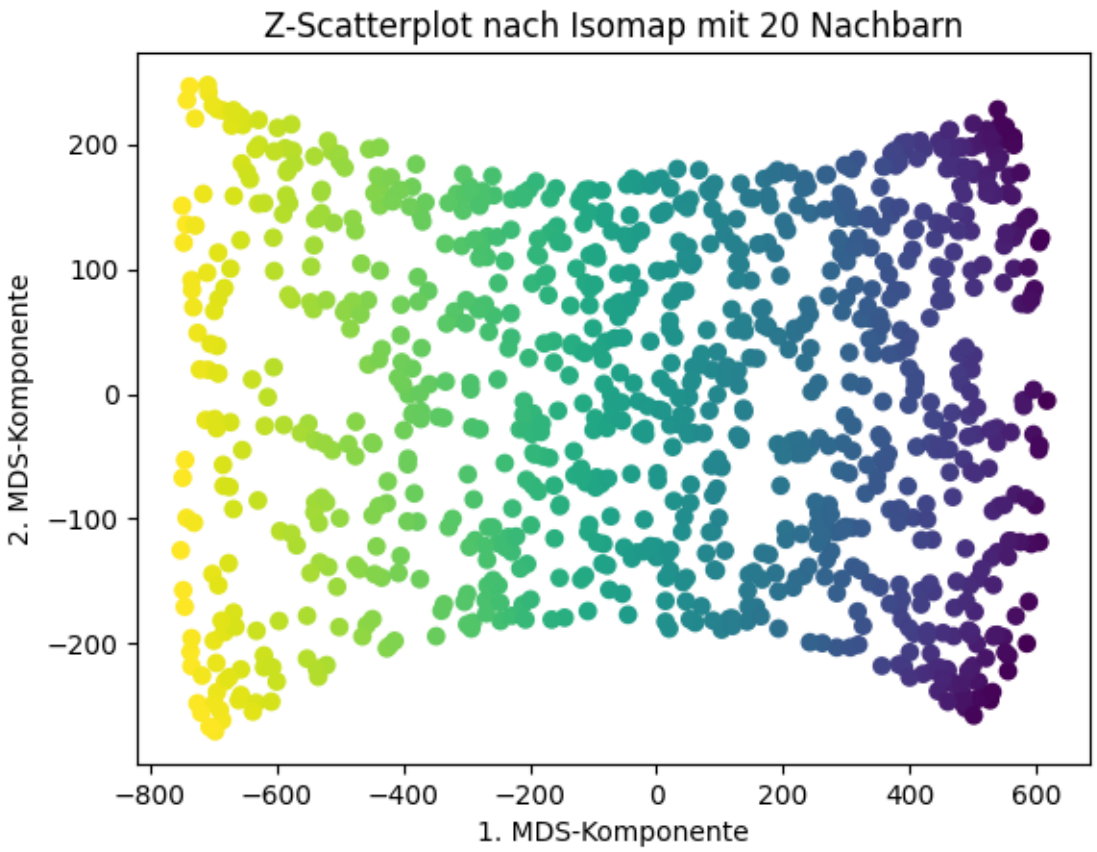
[19]:
```
for K in [7, 8, 9, 10, 20, 30]:
    Y = shortest_path(kneighbors_graph(M, n_neighbors=K, mode='distance'))
    Z, _ = multidimensional_scaling(Y, 2)

    plt.scatter(*Z, c=t)
    plt.title(f"Z-Scatterplot nach Isomap mit {K} Nachbarn")
    plt.xlabel("1. MDS-Komponente")
    plt.ylabel("2. MDS-Komponente")
    plt.show()
```

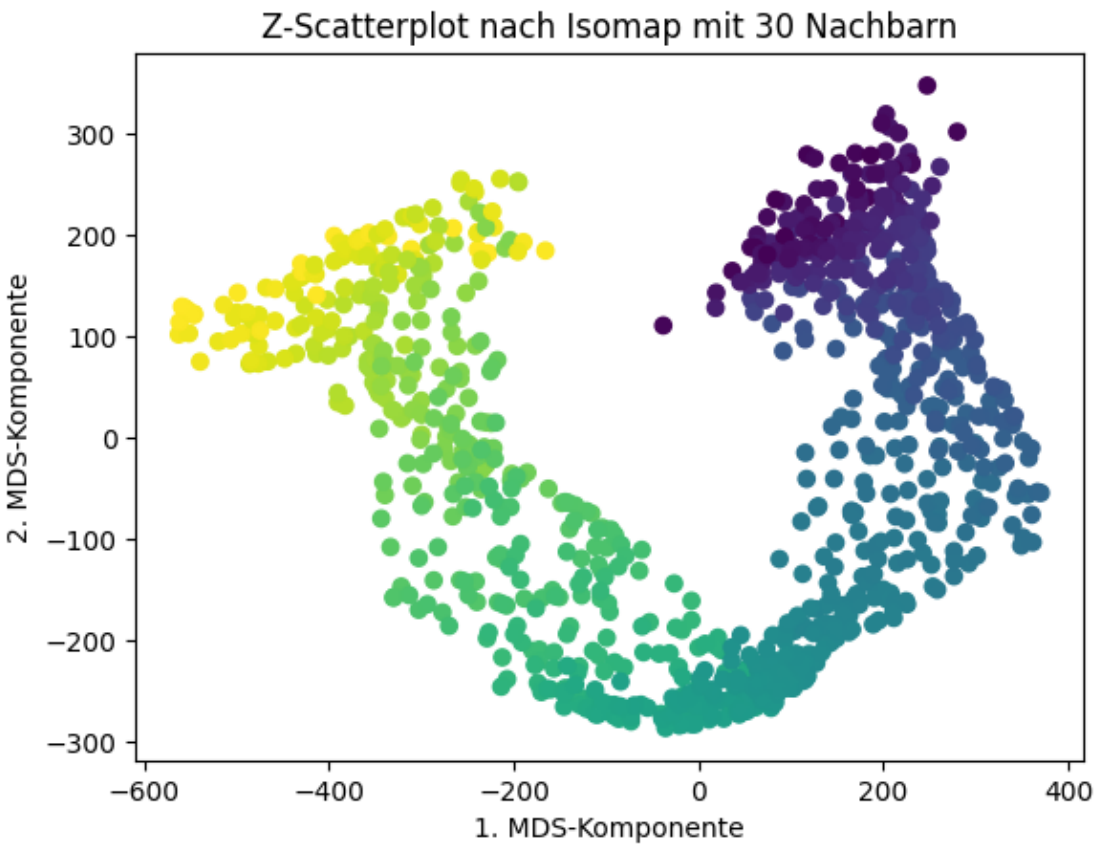

Z-Scatterplot nach Isomap mit 7 Nachbarn
2. MDS-Komponente
1. MDS-Komponente
300
200
100
0
−100
−200
−800
−600
−400
−200
0
200
400
600


Z-Scatterplot nach Isomap mit 8 Nachbarn
300
200
100
0
−100
−200
2. MDS-Komponente
−800
−600
−400
−200
0
200
400
600
1. MDS-Komponente

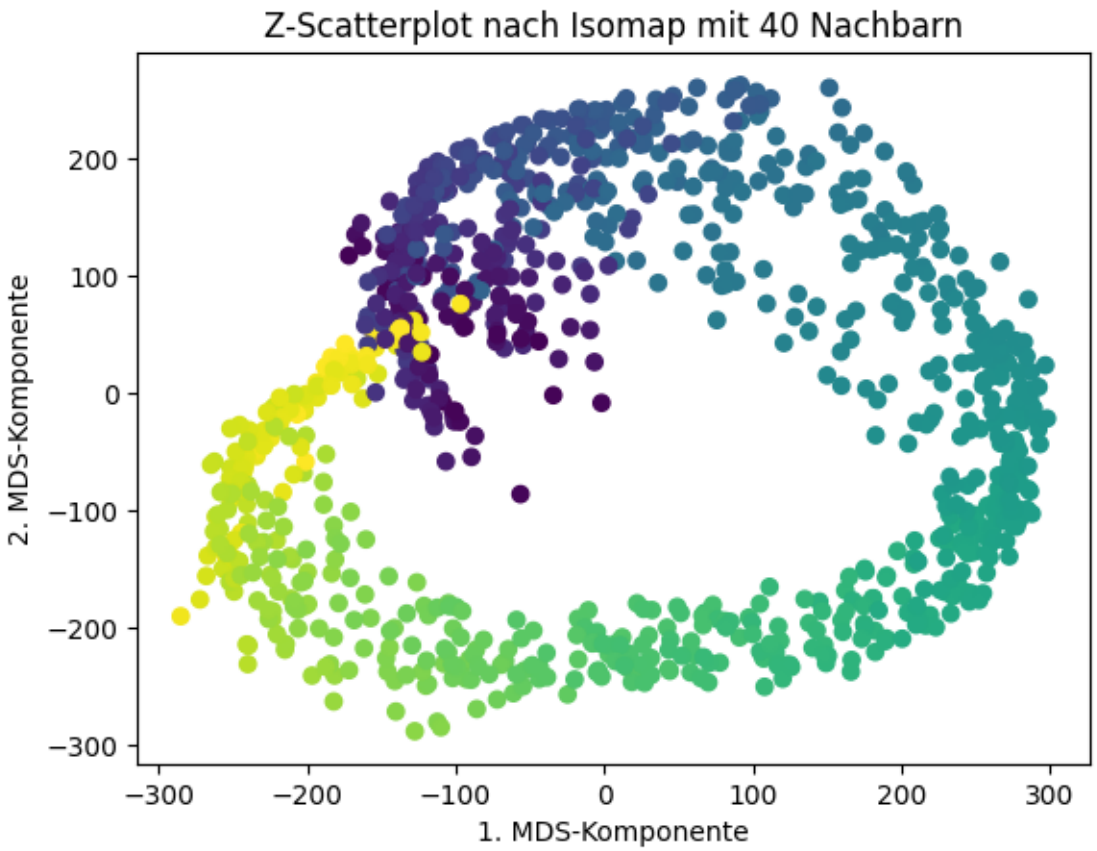

Z-Scatterplot nach Isomap mit 9 Nachbarn
2. MDS-Komponente
1. MDS-Komponente
200
100
0
−100
−200
−300
−600
−400
−200
0
200
400
600
800

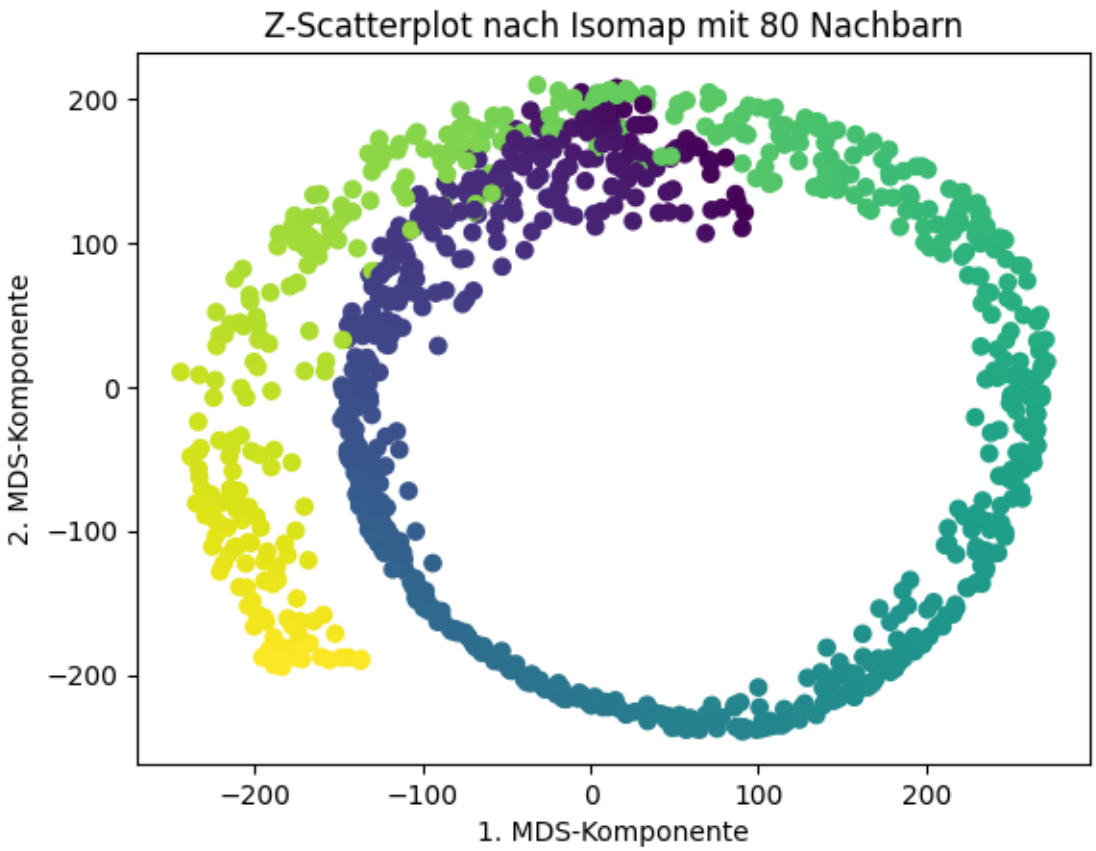

Z-Scatterplot nach Isomap mit 10 Nachbarn
2. MDS-Komponente
1. MDS-Komponente
200
100
0
−100
−200
−600
−400
−200
0
200
400
600
800

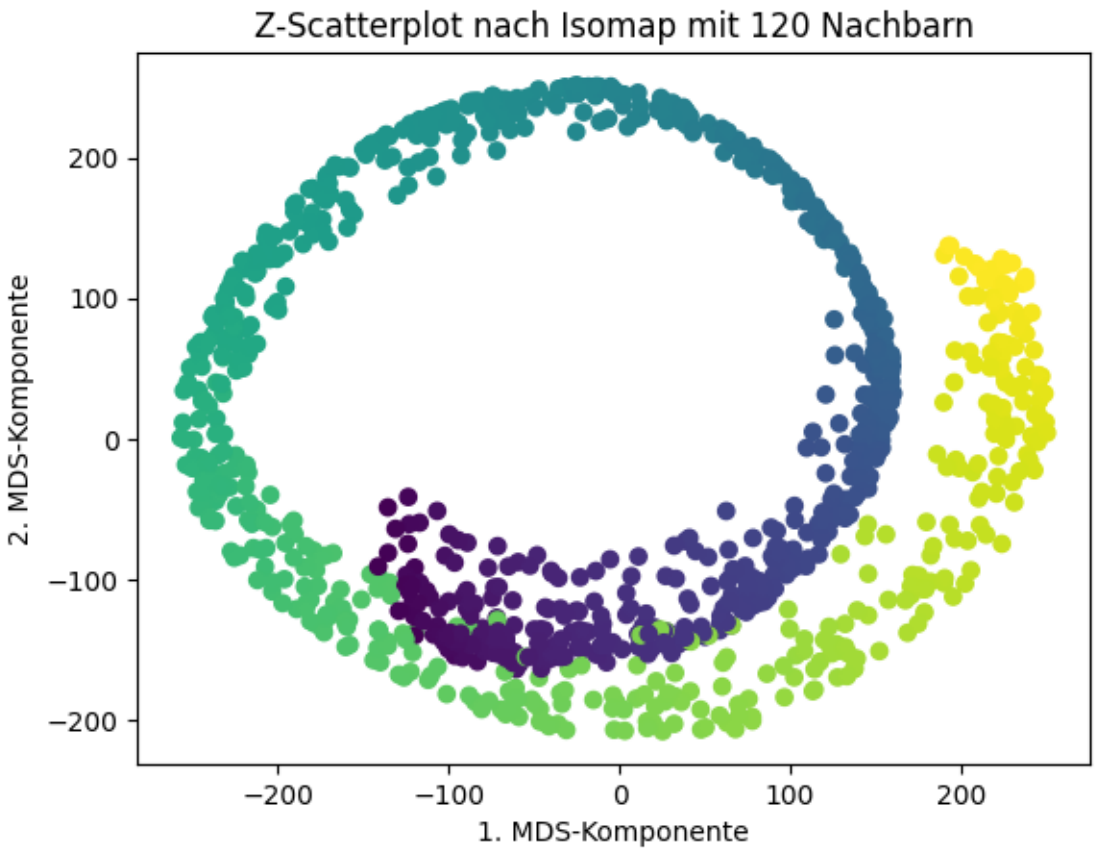

Z-Scatterplot nach Isomap mit 20 Nachbarn
2. MDS-Komponente
1. MDS-Komponente
200
100
0
−100
−200
−800
−600
−400
−200
0
200
400
600

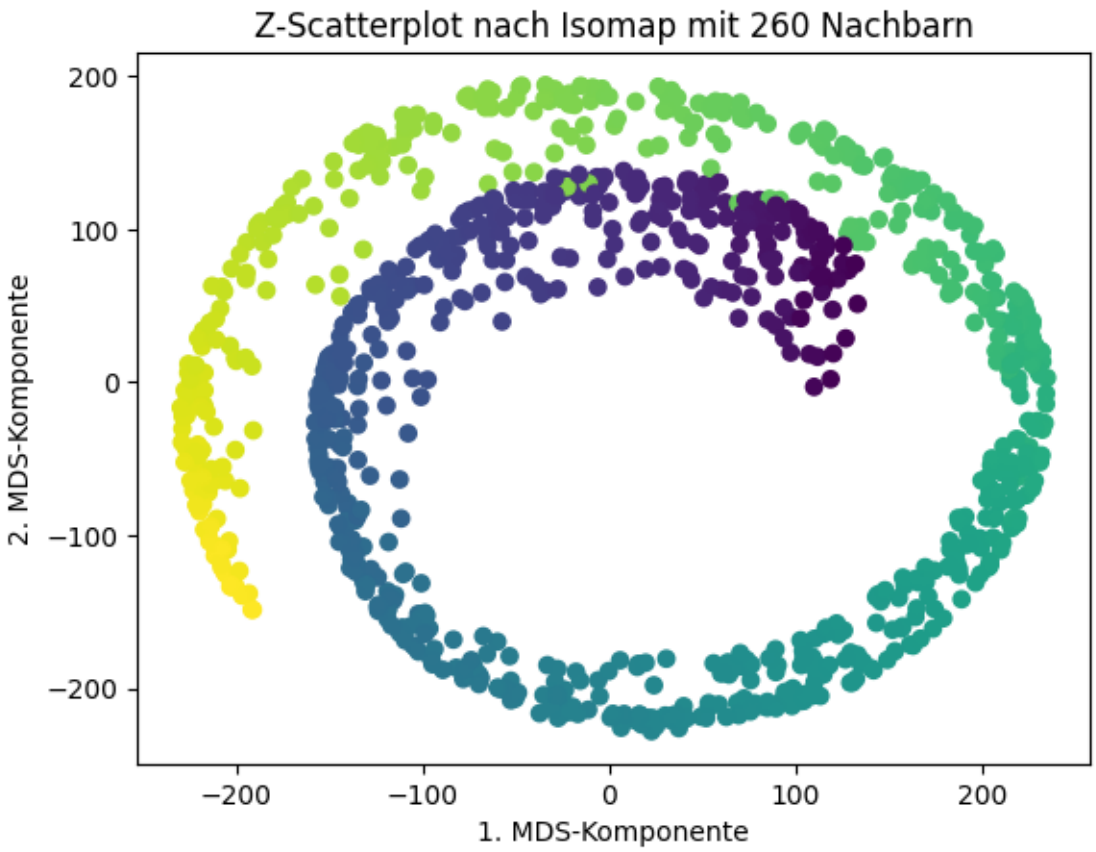


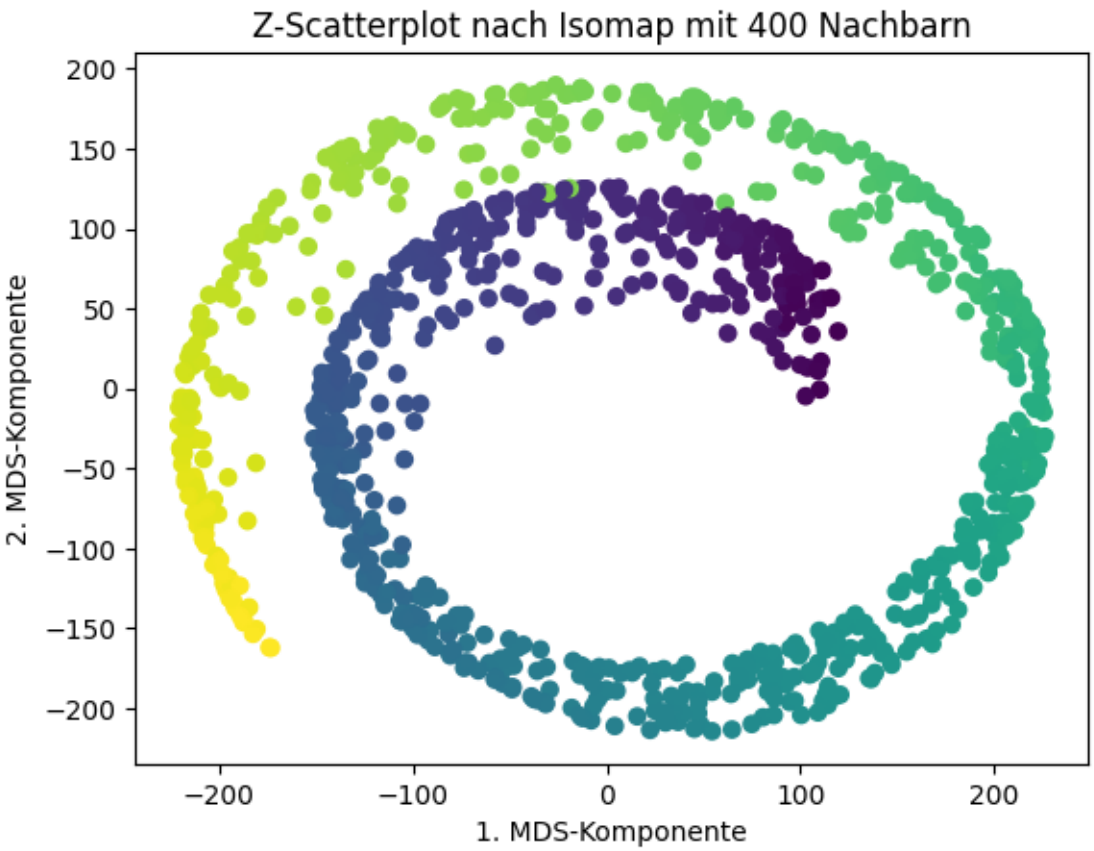
[20]:
```
for K in [40, 80, 120, 260, 400, 600, 800]:
    Y = shortest_path(kneighbors_graph(M, n_neighbors=K, mode='distance'))
    Z, _ = multidimensional_scaling(Y, 2)

    plt.scatter(*Z, c=t)
    plt.title(f"Z-Scatterplot nach Isomap mit {K} Nachbarn")
    plt.xlabel("1. MDS-Komponente")
    plt.ylabel("2. MDS-Komponente")
    plt.show()
```

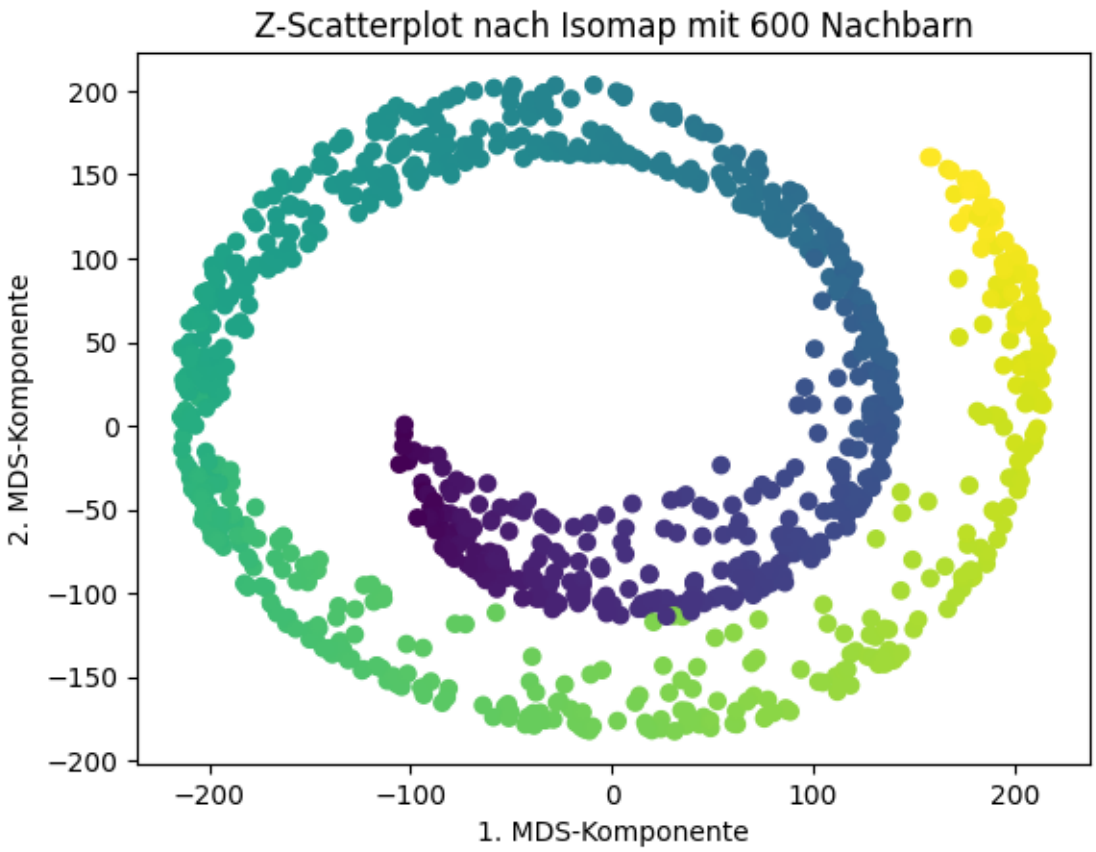

Z-Scatterplot nach Isomap mit 40 Nachbarn
2. MDS-Komponente
1. MDS-Komponente
200
100
0
−100
−200
−300
−300
−200
−100
0
100
200
300

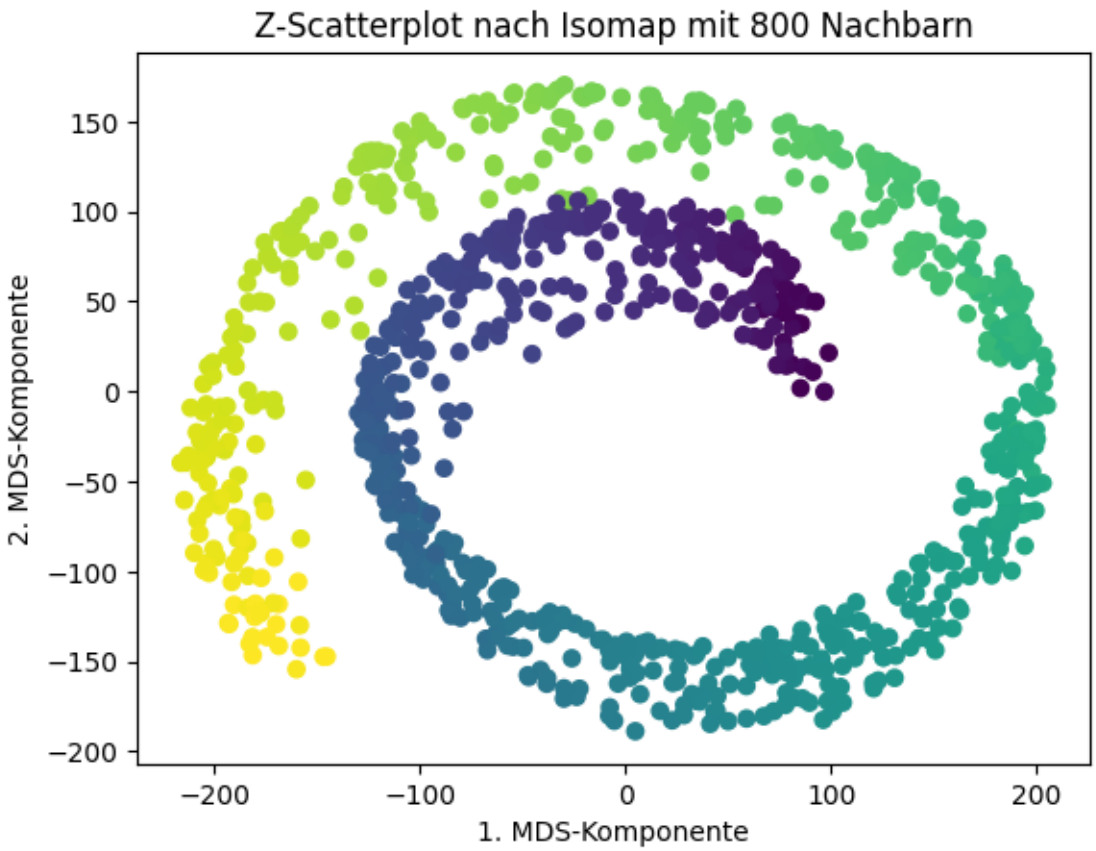

Z-Scatterplot nach Isomap mit 80 Nachbarn
200
100
0
−100
−200
2. MDS-Komponente
−200
−100
0
100
200
1. MDS-Komponente


Z-Scatterplot nach Isomap mit 120 Nachbarn
2. MDS-Komponente
1. MDS-Komponente
200
100
0
−100
−200
−200
−100
0
100
200


Z-Scatterplot nach Isomap mit 260 Nachbarn
200
100
0
−100
−200
2. MDS-Komponente
−200
−100
0
100
200
1. MDS-Komponente


Z-Scatterplot nach Isomap mit 400 Nachbarn
2. MDS-Komponente
1. MDS-Komponente
200
150
100
50
0
−50
−100
−150
−200
−200
−100
0
100
200


Z-Scatterplot nach Isomap mit 600 Nachbarn
2. MDS-Komponente
1. MDS-Komponente
200
150
100
50
0
−50
−100
−150
−200
−200
−100
0
100
200


Z-Scatterplot nach Isomap mit 800 Nachbarn
2. MDS-Komponente
1. MDS-Komponente
150
100
50
0
−50
−100
−150
−200
−200
−100
0
100
200