Freie Universität Berlin

**Prof. R. Rojas**

# Mustererkennung, WS17/18

## Übungsblatt 5

Boyan Hristov, Nedeltscho Petrov

22. November 2017

---

Link zum Git Repository: https://github.com/BoyanH/FU-MachineLearning-17-18/tree/master/Solutions/Homework5

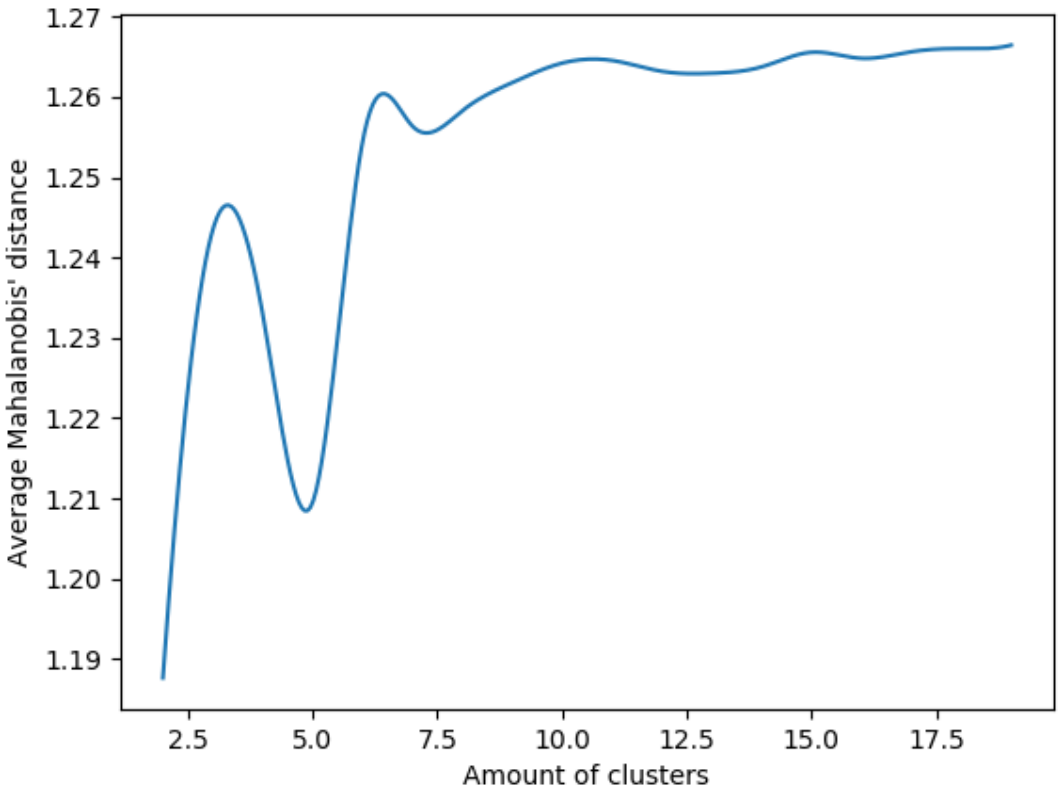
## Expectation Maximization

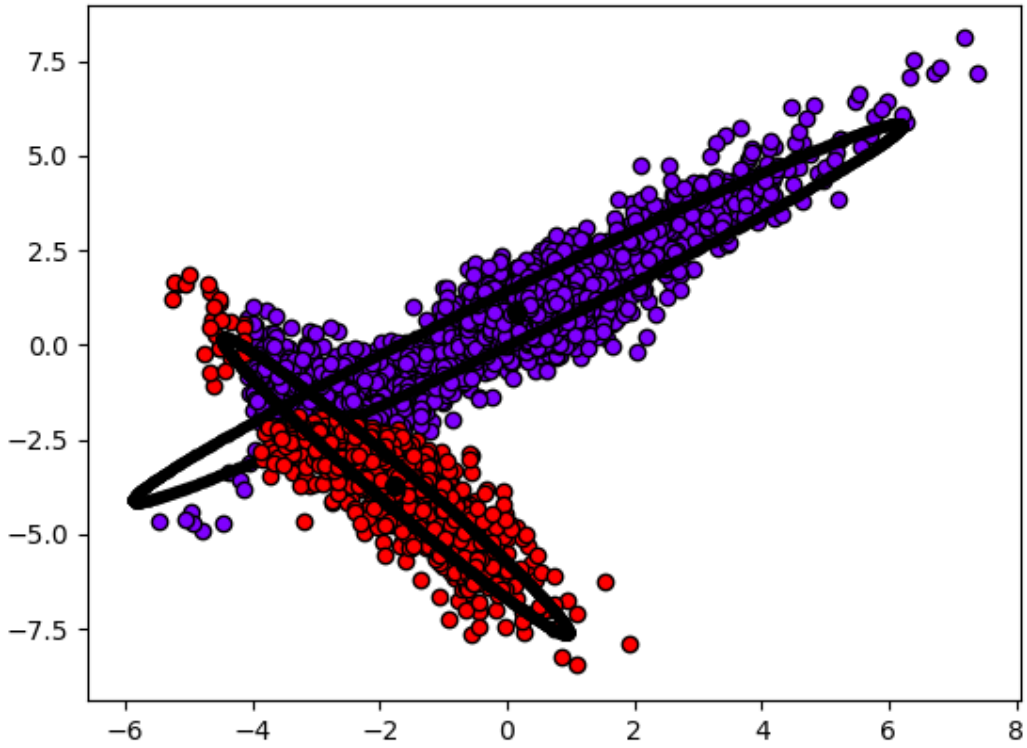
Das Verfahren is für gestreute Datensätze eine deutlich bessere Clustering Methode als K-Means. Viel, womit wir das vergleichen können, haben wir noch nicht gelern. Für den gegebenen Datensatz, der zwei sehr stark ausgeprägte Clusters hat, hat es aber super funktioniert. Wir erkennen aber die Clusters nur anhand der Streuung der Daten, d.h also wir wurden Schwierigkeiten mit K-Means haben, selbe Ergebsnisse zu bekommen.

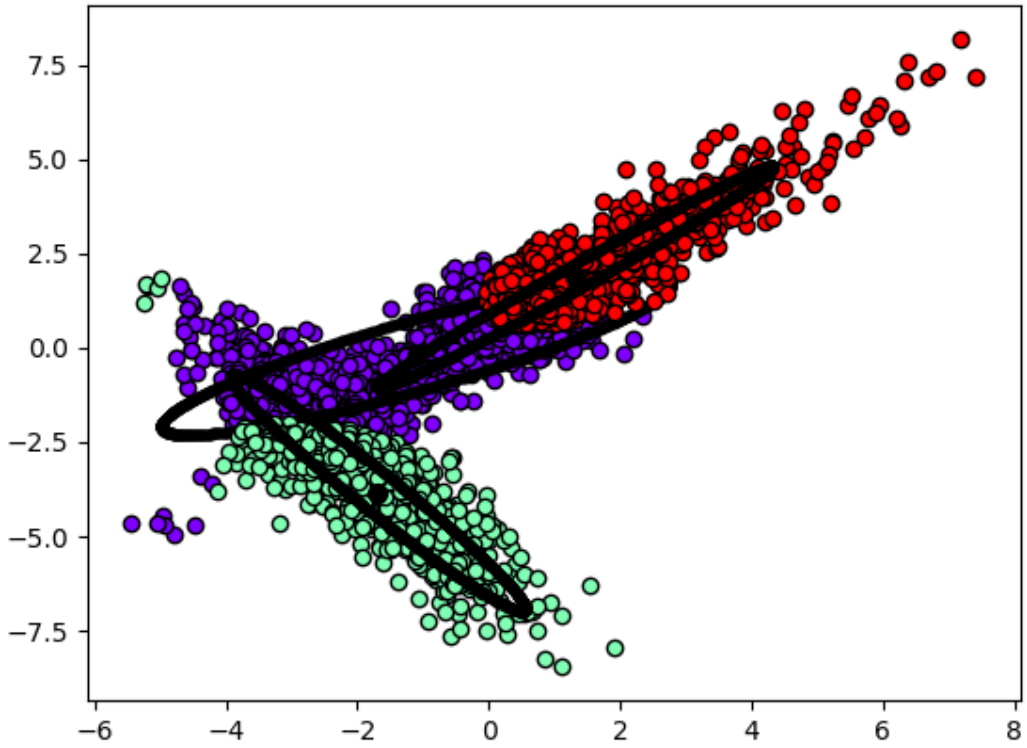
## Plots

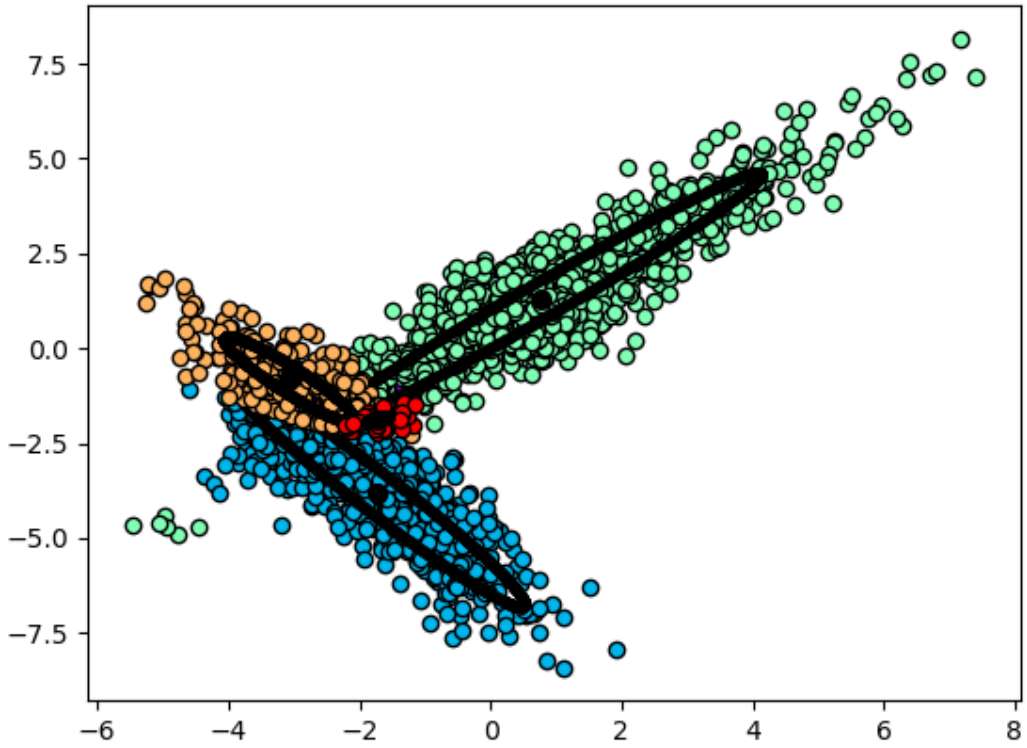
Wir haben das durchscnittliche Distanz zum Cluster (Mahalanobis) von allen Punkten abhängig von den Anzahl der Cluster. Damit die Graphik besser aussieht, haben wir scipy benutz, es existiert aber natürlich kein Clustering mit z.B. 2.3 Cluster.

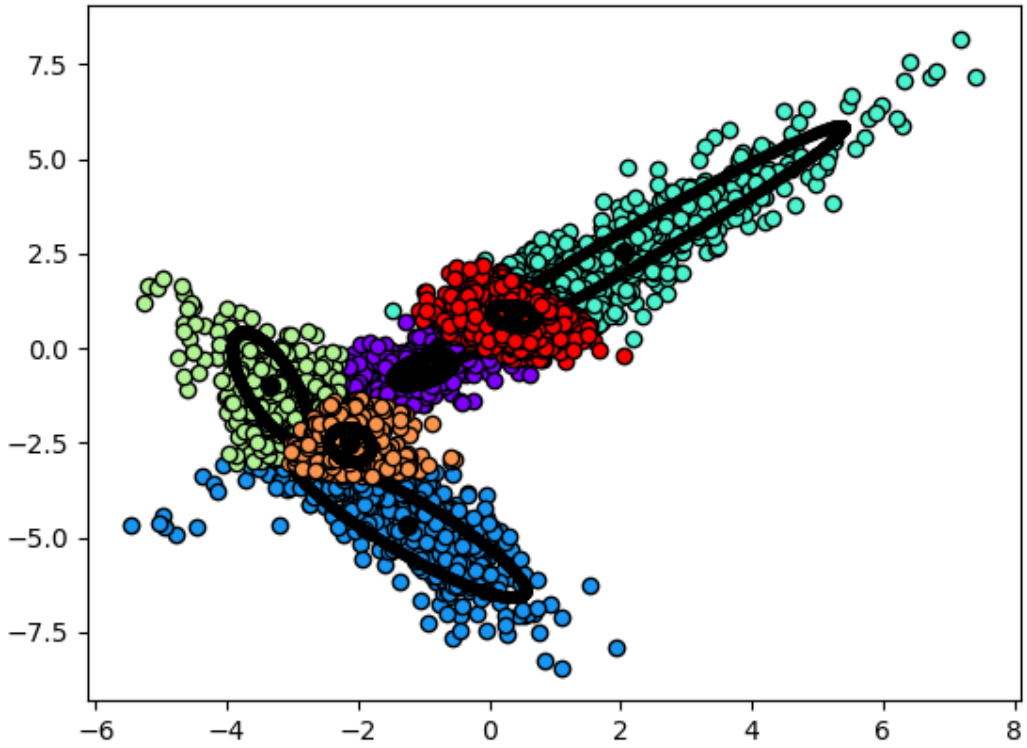
Auf dem Plot kann man die im Tutorium besprochene Ëlbow"leider nicht sehen. Es war nur in einer früheren Version des Programms zu sehen, die ähnlicher zu K-Means war. Es liegt vermutlich daran, dass den Datensatz eine größe und leicht erkennbare Streuung der Daten hat. Nachdem man mehr als 2 Cluster versucht zu finden, werden viele Punkte außerhalb des Streuungsbereichs eines Clusters liegen, aber trotzdem zu dem Cluster gehören. Das ist so, da je mehr man die Daten splitted in mehreren Clusters, desto weniger wichtig wird die Streuung eines Clusters verglichen mit einem anderen. Naja, "long story short"wird denken 2 Clusters sind am bestens geeignet für diesen Datensatz.

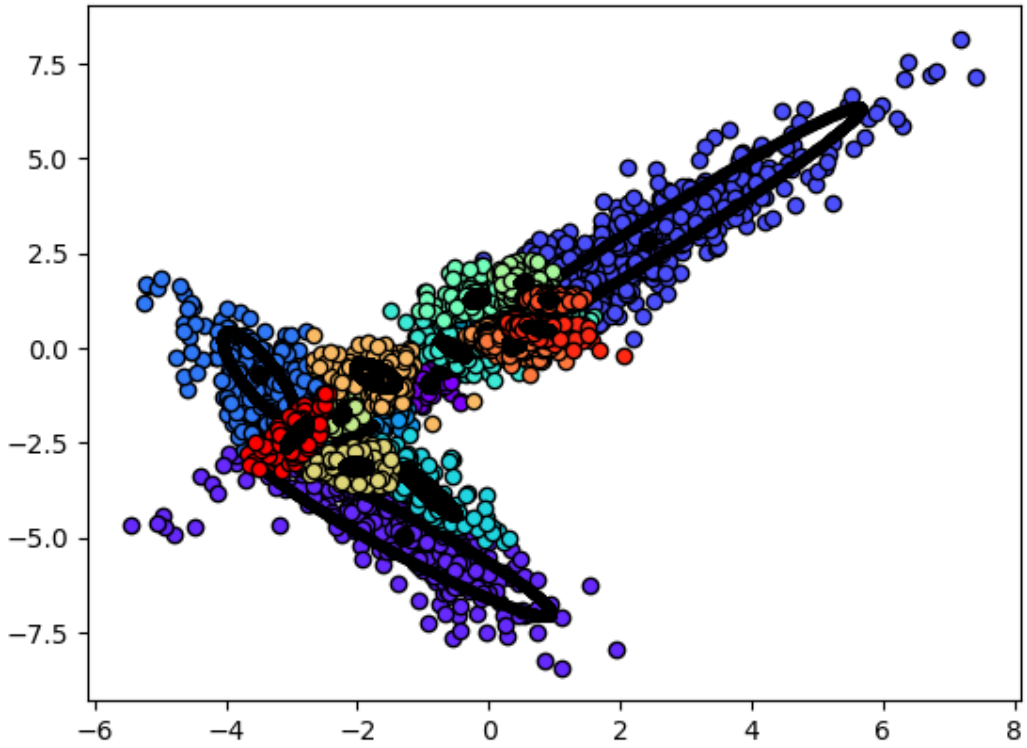

1.27
1.26
1.25
1.24
1.23
1.22
1.21
1.20
1.19
Average Mahalanobis' distance
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Amount of clusters


7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−6
−4
−2
0
2
4
6
8


7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−6
−4
−2
0
2
4
6
8


7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−6
−4
−2
0
2
4
6
8


7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−6
−4
−2
0
2
4
6
8




## Details zur Implementierung

Interessant für die Implementierung sind wahrscheinlich die Berechnung von dem Abstand und die Entscheidung, wann der Algorithmus eigentlich fertig ist.

Mahalanobis Abstand - wir könnten uns hier den Wurzel sparen, so haben wir aber deutlich präziseren durchschnittlichen Abstand bekommen, der besser zu plotten war. Da sonst die Großteil der Implementierung vektorisiert ist, hatten wir nicht zu viele Sorgen wegen Performance. Sehr interessant ist es eigentlich nicht, nur die Formel aus der Vorlesung..

```
    def get_distance_mahalanobis_to_cluster(self, x, i_cluster):
        # the square root could be removed, but helps for better plotting
        return math.sqrt((x - self.cluster_centers[i_cluster]).dot(
            self.inv_covariances_per_cluster[i_cluster]).dot((x - self.cluster_centers[
    i_cluster]).T))
```

Cluster Zentren berechnen, vektorisiert

```
    def calculate_cluster_centers(self):
        get_cluster_centers = np.vectorize(lambda x, points_per_center: points_per_center[x
    ].mean(0),
                                           signature='(),(m)->(n)')
        self.cluster_centers = get_cluster_centers(self.cluster_indexes, self.
    points_per_cluster)
```

Berechnung von Kovarianzmatrizen - dafür haben wir Numpy benutzt, interessant es aber der Fall, wenn wir nicht genug Punkte haben. Dann nehmen wir bloß die Identitätsmatrix. ägainßteht im Kommentare, da wir bei der Initialisierung auch die Identitätsmatrix nehmen.

```
    def calculate_covariances(self):
        # if we only have the center in our cluster (empty cluster) then just use the identity
        # matrix as covariance matrix again
        get_covariances = np.vectorize(lambda x, points_for_cluster: np.cov(points_for_cluster[x],
                                       rowvar=False, bias=True) if len(points_for_cluster[x]) > 1 else
                                       np.identity(len(points_for_cluster[x][0])), signature='(),(m)->(n,n)')
        self.covariances_per_cluster = get_covariances(self.cluster_indexes, self.points_per_cluster)
        self.inv_covariances_per_cluster = np.vectorize(lambda x: np.linalg.pinv(x),
                                                        signature='(m,n)->(m,n)')(self.covariances_per_cluster)
```

”Main”Methode - wir haben hier anhand von Flags zwie Vorgehensweisen implementiert. Entweder terminiert man, wenn die Clusterzentren sich nicht so viel bewegen, oder falls man 1. schlechtere Ergebnisse bekommt oder 2. ein Wunschergebniss erreicht hat. In beiden Implementierungen gibt es eine maximale Anzahl von Iterationen (30, vermutlich wäre 10 besser, aber... i

```
    def apply_expectation_maximization(self, k=0):
        if VERBOSE:
            print('iteration: {}'.format(k))
        old_centers = np.copy(self.cluster_centers)
        old_distance = np.copy(self.mean_distance)
        self.reset_points_per_cluster()
        self.assign_points_to_clusters()
        self.calculate_cluster_centers()
        self.calculate_covariances()
        self.update_mean_distance_to_cluster_centers()

        # for some reason old_distance is None or old_distance > ...
        # was throwing errors cannot compare NoneType with int
        # therefore the less readable not old_distance :X

        # but basically, if judging on distance for when to stop,
        # if the average distance gets worse, we reach the max amount of iterations or we reach our desired
        # threshold, stop
        if USE_DISTANCE_THRESHOLD:
            if (self.mean_distance > DISTANCE_THRESHOLD and
                        k < self.max_iterations and (not old_distance or old_distance > self.mean_distance)):
                self.apply_expectation_maximization(k + 1)

        # other method to determine when to stop is by simply checking if the cluster centers still move enough
        elif abs((old_centers - self.cluster_centers).sum()) > MOVEMENT_THRESHOLD and k < self.max_iterations:
            self.apply_expectation_maximization(k + 1)
```

# Vollständiges Code zu Expectation Maximization

```
from Parser import parse_data
import numpy as np
from Helpers import save_plot, plot_covariance
from random import random
```

```
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from scipy.interpolate import spline
import random
import math

CHOOSE_INITIAL_CENTERS_RANDOMLY = True
USE_DISTANCE_THRESHOLD = True
MOVEMENT_THRESHOLD = 0.002
DISTANCE_THRESHOLD = 0.01

VERBOSE = False
PLOT_MEAN_FOR_CLUSTERS_COUNT = True
PLOT_CLUSTERING_FOR_SOME_K = True
SAVE_PLOTS = True


class ExpectationMaximization:
    @staticmethod
    def get_initial_centers_from_data_set(data, k):
        if CHOOSE_INITIAL_CENTERS_RANDOMLY:
            random.seed(8)
            return np.array(random.choices(data, k=k), dtype=np.float64)

        min_point = data.min(0)
        max_point = data.max(0)
        centers = []

        for i in range(k):
            centers.append(min_point + (max_point - min_point) / k)

        return centers

    def __init__(self):
        self.data = None
        self.k_clusters = None
        self.sigma = None
        self.cluster_centers = None
        self.points_per_cluster = None
        self.inv_covariances_per_cluster = []
        self.covariances_per_cluster = []
        self.cluster_indexes = None
        self.last_diff = None
        self.mean_distance = None
        self.max_iterations = None

    def reset_points_per_cluster(self):
        self.points_per_cluster = [[x] for x in self.cluster_centers]

    def cluster(self, data, k_clusters, max_iterations=30):
        self.data = data
        self.mean_distance = None
        self.last_diff = None
        self.k_clusters = k_clusters
        self.cluster_indexes = [x for x in range(self.k_clusters)]
        self.cluster_centers = ExpectationMaximization.get_initial_centers_from_data_set(
    data, k_clusters)
        self.covariances_per_cluster = [[np.identity(len(x))] for x in self.cluster_centers
    ]
        self.inv_covariances_per_cluster = self.covariances_per_cluster
        self.reset_points_per_cluster()
        self.max_iterations = max_iterations
        self.apply_expectation_maximization()

    def apply_expectation_maximization(self, k=0):
        if VERBOSE:
            print('iteration: {}'.format(k))
        old_centers = np.copy(self.cluster_centers)
```

```
71          old_distance = np.copy(self.mean_distance)
72          self.reset_points_per_cluster()
73          self.assign_points_to_clusters()
74          self.calculate_cluster_centers()
75          self.calculate_covariances()
76          self.update_mean_distance_to_cluster_centers()

78          # for some reason old_distance is None or old_distance > ...
79          # was throwing errors cannot compare NoneType with int
80          # therefore the less readable not old_distance :X

82          # but basically, if judging on distance for when to stop,
83          # if the average distance gets worse, we reach the max amount of iterations or we
    reach our desired
84          # threshold, stop
85          if USE_DISTANCE_THRESHOLD:
86              if (self.mean_distance > DISTANCE_THRESHOLD and
87                      k < self.max_iterations and (not old_distance or old_distance >
    self.mean_distance)):
88                  self.apply_expectation_maximization(k + 1)

90          # other method to determine when to stop is by simply checking if the cluster
    centers still move enough
91          elif abs((old_centers - self.cluster_centers).sum()) > MOVEMENT_THRESHOLD and k <
    self.max_iterations:
92              self.apply_expectation_maximization(k + 1)

94      def update_mean_distance_to_cluster_centers(self):
95          # 1. get distance for every point in each cluster in a single array
96          # 2. get mean of that

98          dis_for_x_in_k = np.vectorize(lambda x, i: self.get_distance_mahalanobis_to_cluster
    (x, i),
99                                        signature='(m),()->()')

101         # didn't manage to vectorize this one ;/ somehow, numpy doens't like jagged arrays
102         distances_for_points = list(map(lambda x: dis_for_x_in_k(self.points_per_cluster[x
    ], x),
103                                         self.cluster_indexes))

105         flattened_distances = []
106         for cluster_distances in distances_for_points:
107             flattened_distances = flattened_distances + list(cluster_distances)

109         self.mean_distance = np.array(flattened_distances, dtype=np.float64).mean()

111         if VERBOSE:
112             print('Average Mahalanobis\' distance to cluster center: {}'.format(self.
    mean_distance))

114     def assign_points_to_clusters(self):
115         assign_points_to_clusters = np.vectorize(lambda x: self.assign_point_to_cluster(x),
116                                                  signature='(m)->()')
117         assign_points_to_clusters(self.data)

119         for idx, list in enumerate(self.points_per_cluster):
120             self.points_per_cluster[idx] = np.array(self.points_per_cluster[idx])

122     def assign_point_to_cluster(self, point):
123         distances_to_clusters = [self.get_distance_mahalanobis_to_cluster(point, i) for i
    in self.cluster_indexes]
124         closest_cluster_idx = np.argmin(distances_to_clusters)
125         self.points_per_cluster[closest_cluster_idx].append(point)

127     def get_distance_mahalanobis_to_cluster(self, x, i_cluster):
128         # the square root could be removed, but helps for better plotting
129         return math.sqrt((x - self.cluster_centers[i_cluster]).dot(
```

```
            self.inv_covariances_per_cluster[i_cluster]).dot((x - self.cluster_centers[i_cluster]).T))

    def calculate_cluster_centers(self):
        get_cluster_centers = np.vectorize(lambda x, points_per_center: points_per_center[x].mean(0),
                                           signature='(),(m)->(n)')
        self.cluster_centers = get_cluster_centers(self.cluster_indexes, self.points_per_cluster)

    def calculate_covariances(self):
        # if we only have the center in our cluster (empty cluster) then just use the identity
        # matrix as covariance matrix again
        get_covariances = np.vectorize(lambda x, points_for_cluster: np.cov(points_for_cluster[x],
                                       rowvar=False, bias=True) if len(points_for_cluster[x]) > 1 else
                                       np.identity(len(points_for_cluster[x][0])), signature='(),(m)->(n,n)')
        self.covariances_per_cluster = get_covariances(self.cluster_indexes, self.points_per_cluster)
        self.inv_covariances_per_cluster = np.vectorize(lambda x: np.linalg.pinv(x),
                                                        signature='(m,n)->(m,n)')(self.covariances_per_cluster)


data = parse_data()
em = ExpectationMaximization()

# Depending on the cluster centers that are chosen, results differ
# Though mostly between 3 and 4 clusters fit best for the current data set
if PLOT_MEAN_FOR_CLUSTERS_COUNT:
    cluster_count_experiments = [x for x in range(2, 20)]
    cluster_count_mean_distance_results = []

    for cluster_count in cluster_count_experiments:
        em.cluster(data, cluster_count)
        cluster_count_mean_distance_results.append(np.copy(em.mean_distance))

        if VERBOSE:
            for idx, points in enumerate(em.points_per_cluster):
                print('Points in cluster #{}: {}'.format(idx, len(points)))

    x = np.linspace(min(cluster_count_experiments), max(cluster_count_experiments), 300)
    y = spline(cluster_count_experiments, cluster_count_mean_distance_results, x)

    figure = plt.figure()
    plt.plot(x, y)
    plt.xlabel('Amount of clusters')
    plt.ylabel('Average Mahalanobis\' distance')

    if SAVE_PLOTS:
        save_plot(figure, './plots/avrg_distance_for_k.png')
    else:
        plt.show()

if PLOT_CLUSTERING_FOR_SOME_K:
    plot_for_k_s = [2,3,5,6, 20]
    # plot_for_k_s = [2]

    for k in plot_for_k_s:
        em.cluster(data, k)
        colors = cm.rainbow(np.linspace(0, 1, k))

        fig = plt.figure()
        ax1 = fig.add_subplot(111)
```

```
        for cl_idx in em.cluster_indexes:
            X = em.points_per_cluster[cl_idx]
            x, y = zip(*X)
            # ax1.figure(figsize=(15, 10))
            ax1.scatter(x, y, edgecolors="black", c=colors[cl_idx])

            center = em.cluster_centers[cl_idx]
            covariance = em.covariances_per_cluster[cl_idx]
            plot_covariance(ax1, center[0], center[1], covariance)

        if SAVE_PLOTS:
            save_plot(fig, './plots/plot_for_k_{}.png'.format(k))
        else:
            plt.show()
```

```
import csv
import numpy as np
import os
from sklearn.model_selection import train_test_split


def parse_data():
    file_name = os.path.join(os.path.dirname(__file__), './Dataset/2d-em.csv')
    csv_file = open(file_name, 'rt')
    reader = csv.reader(csv_file, delimiter=',', quoting=csv.QUOTE_NONE)

    return np.array([row for row in reader], dtype=np.float64)
```

```
import os
import pandas as pd
from numpy import pi, sin, cos
import numpy as np
import matplotlib.pyplot as plt

RGB_BLACK = [0, 0, 0]


def save_plot(fig, path):
    fig.savefig(os.path.join(os.path.dirname(__file__), path))


def plot_covariance(ax1, x_initial, y_initial, cov):
    num_points = 1000
    radius = 1.5  # adjusted radius, seems more correct this way

    # plot a circle
    arcs = np.linspace(0, 2 * pi, num_points)
    x = radius * sin(arcs)
    y = radius * cos(arcs)

    # stretch it according to the covariance matrix
    xy = np.array(list(zip(x, y)))
    x, y = zip(*xy.dot(cov))

    # move it in the space so it's center is above the cluster's center
    x = x + x_initial
    y = y + y_initial
```

```
258     ax1.scatter(x, y, c=RGB_BLACK, s=10)  # plot covariance
259     ax1.scatter([x_initial], [y_initial], c=RGB_BLACK, s=50)  # plot center
```