CS564 Foundations of Machine Learning
Assignment: 3

Mukuntha N S (1601CS27)
R J Srivatsa (1601CS35)
Shikhar Jaiswal (1601CS44)

September 18, 2019

1 Assignment Description

Finding the correct cluster centers is one of the important steps for getting good results in clustering. We try to find the correct number of clusters for the **Diabetes Dataset** .

2 Procedure

Installation

Install the following dependencies either using pip or through conda in a Python 3.5+ environment:
• jupyter
• numpy
• pandas

```
python3 -m pip install jupyter numpy pandas
```

or alternatively,

```
conda install -c anaconda jupyter numpy pandas
```

Running The Notebook

To run the program, head to the directory $Assignment3/Q1$. Please note that the **dataset should be present in the same directory** as the jupyter notebook. Use the following command to run the notebook:

```
jupyter notebook Q1.ipynb
```

3 Discussion

The primary objective of the assignment is to calculate the the number of cluster centers while running DBSCAN algorithm, and plot the core and noise points. The following sub-sections contain the explanation for individual code snippets. For a detailed look at the output, please refer the notebook and the individual files provided.

Notebook Code

Pre-processing and Visualization

Import the Python dependencies, and check the data samples and their values.

```
# Import the required libraries.
import re
import math
import random
import collections
import numpy as np
import pandas as pd


# Set the random seed.
random.seed(15)
np.random.seed(15)

data = pd.read_csv('diabetes.arff', header = None)
# Drop the last column.
data = data.drop(data.columns[8], axis = 1)

print(data)
```

Part A

We normalize all data and run the DBSCAN algorithm with epsilon parameter set at 2 and minimum number of samples set at 5. Finally, we store the count of the number of clusters observed.

```
# Normalize the dataframe.
X = data.apply(lambda column: (column - column.mean()) / column.std(), axis = 0)

# Function for finding the set of epsilon-neighbourhood points for a given point.
def region_query(point, eps, data):
    neighbour_points = []

    for index, row in data.iterrows():
        if np.linalg.norm(point - row) <= eps:
            neighbour_points.append((index, row))

    return neighbour_points
```

```
# Function for expanding the current cluster size by including more number
# of points in the cluster.
def expand_cluster(data, neighbour_points, label, labels, eps, min_samples):
    i = 0

    while i < len(neighbour_points):
        index = neighbour_points[i][0]
        point = neighbour_points[i][1]

        if labels[index] == -1:
            # In case the current point is labelled as noise, re-label
            # it to boundary point.
            labels[index] = label - 1
        elif labels[index] == 0:
            # In case the current point is unvisited, re-label it to
            # core point and add its neighbour points to the current
            # queue.
            new_neighbour_points = region_query(point, eps, data)

            if len(new_neighbour_points) >= min_samples:
                labels[index] = label
                neighbour_points = neighbour_points + new_neighbour_points
            else:
                labels[index] = label - 1

        i += 1

# Function for implementing the DBSCAN algorithm.
def DBSCAN(data, eps, min_samples):
    # Initialize a list keeping track of visited points.
    labels = [0] * data.shape[0]
    label = 0

    for index, row in data.iterrows():
        if labels[index] != 0:
            continue

        # Generate the epsilon neighbourhood of the point.
        neighbour_points = region_query(row, eps, data)

        if len(neighbour_points) < min_samples:
            # In case of lesser number of samples in the
            # epsilon neighbourhood, label the point as noise.
            labels[index] = -1
        else:
            # Increment the label by two, where even label
            # signifies core point and odd label signifies
            # boundary point.
```

```
            label += 2
            labels[index] = label
            expand_cluster(data, neighbour_points, label, labels, eps, min_samples)

    return labels

labels = DBSCAN(X, eps = 2, min_samples = 5)

# Number of clusters in labels, ignoring noise if present.
n_clusters = max(labels) // 2

print('Estimated Number of Clusters: ', n_clusters)
```

```
Output:
Estimated Number of Clusters: 3
```

Part B

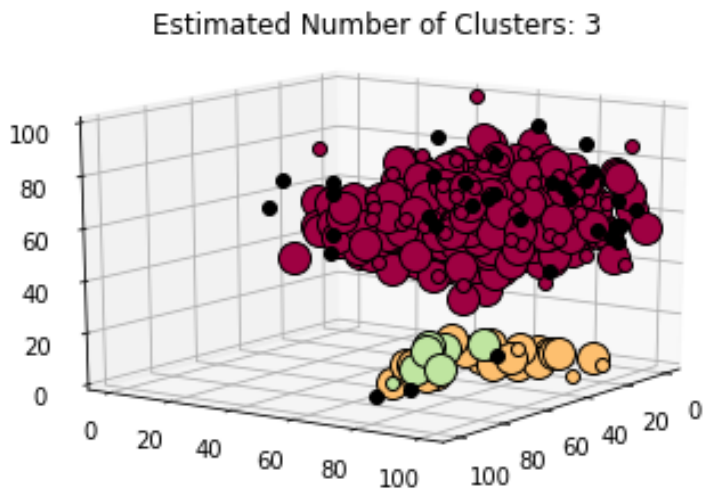
We finally store the core points belonging to individual clusters and the associated boundary points in separate files.

```
# Collect the indices of data pertaining to different clusters and noise.
noiseindices = [index for index, label in enumerate(labels) if label == -1]
boundary1indices = [index for index, label in enumerate(labels) if label == 1]
core1indices = [index for index, label in enumerate(labels) if label == 2]
boundary2indices = [index for index, label in enumerate(labels) if label == 3]
core2indices = [index for index, label in enumerate(labels) if label == 4]
boundary3indices = [index for index, label in enumerate(labels) if label == 5]
core3indices = [index for index, label in enumerate(labels) if label == 6]

# Segregate the data into appropriate dataframes.
noise = data.iloc[noiseindices, :]
boundary1 = data.iloc[boundary1indices, :]
core1 = data.iloc[core1indices, :]
boundary2 = data.iloc[boundary2indices, :]
core2 = data.iloc[core2indices, :]
boundary3 = data.iloc[boundary3indices, :]
core3 = data.iloc[core3indices, :]

# Write the files to output.
noise.to_csv('noise_data.csv')
boundary1.to_csv('boundary_data_cluster_1.csv')
core1.to_csv('core_data_cluster_1.csv')
boundary2.to_csv('boundary_data_cluster_2.csv')
core2.to_csv('core_data_cluster_2.csv')
boundary3.to_csv('boundary_data_cluster_3.csv')
core3.to_csv('core_data_cluster_3.csv')
```

Plot



Additionally please refer $Q1/core\_data\_cluster\_1.csv$ and similar files for the output.