Aprendizagem Automática

Assignment 2 Report

Bruno Fernandes Nº 42200

Pedro Trindade Nº 41661

Paulo Martins Nº 41982

Introduction

For this assignment, our goal was to cluster all the seismic events with a magnitude of at least 6.5 that happened in the last 100 years.

To do this, we were asked to parametrize and compare the **K-Means** and **DBSCAN** clustering algorithms, exploring multiple combinations of parameters for both of them while using the silhouette score to estimate the quality of all the obtained clusters.

General Overview

K-Means Algorithm

The **K-Means** clustering algorithm consists in dividing the data set into *k* clusters (*k* being a predefined number). The idea of this algorithm is to define *k* synthetic points (one per cluster) named centroids that represent the mean value of all the points that belong to the cluster. So, clusters are defined by the proximity of each point to the mean of the clusters, each point belonging to the cluster represented by the closest centroid. We can also say that **K-Means** is an *exclusive* (each points belongs only to one cluster), *partitional* (clusters are created at the same level), *complete* (all points belong to a cluster) and *prototype-based* (we determine the prototypes or centroids and then cluster according to these prototypes) clustering method.

This algorithm works well for globular clusters with similar dispersion, with the correct *k* value. So, we can predict that it will not be very adequate for our problem, as our data clusters don't seem to be very globular, since they represent the seismic events, often associated as seismic ridges.

DBSCAN Algorithm

The **DBSCAN** is a density-based spatial clustering algorithm of applications with noise, which takes a different approach from the **K-Means** algorithm and solves some of its problems, namely the problem of the shape of the clusters, since cluster membership propagates along the paths of nearby core points. So, basically this algorithm tries to define clusters based on the local density of points. (number of points within a specified radius). To do this we need to define two important parameters:

Eps – maximum radius of the neighborhood

MinPts – minimum number of points in a Eps-neighborhood of a point

We can say that a point *p* is a core point if it has at least *MinPts* within *Eps*; *p* is a border point if it has fewer than *MinPts* within *Eps* but is in the neighborhood of a core point; otherwise *p* is a noise point.

A point *q* is reachable from point *p* if *q* is directly reachable from *p* (its in the neighborhood of *p*) or if its in the neighborhood of a core point that is directly reachable from *p*. Reachable points are going to be clustered (*q* is reachable from *p* if there is a path of reachable core points from *p* to *q*): if *p* is a core point a cluster is created for *p* and all the neighbors of *p* are added to the cluster, if any neighbor of *p* is a core point belonging to another cluster, the clusters are merged.

We predict the **DBSCAN** algorithm is predicted to be adequate for our problem has it has a good resistance to noise and it can handle clusters of different shapes and sizes. The weak points of the algorithm are highly varying densities and high-dimensional data, which is not the case in our data. The bigger problem is that **DBSCAN** is highly sensitive to its parameters (*Eps* and *MinPts*) so we have to correctly find the best parameters in order to obtain good results.

Silhouette Score

The silhouette score is a form of unsupervised clustering validation. Given a point x, the silhouette score of point x (s(x)) is:

$$s(x) = \frac{b(x)-a(x)}{\max(a(x),\ b(x))}$$

a(x) – average distance of x to all the points in the same cluster

b(x) – average distance of x to all points in the nearest cluster (lowest average distance)

So, (x) values range from -1 to 1. If s(x) = -1, the results are very bad, because the point x will be much closer to the points in the nearest cluster than to points in its own cluster. If s(x) = 1, the results are very good, because the point x will be really close to the points in its cluster. So we want a(x) to be much smaller than b(x).

Basically the silhouette score is an adequate measure for the cases where the clusters are globular. So, it doesn't work well in the situations where the K-Means algorithm also doesn't work well, as it does not take into account different dispersions in the clusters. In our case, the silhouette score is not a very adequate measure, because the clusters are clearly not globular, so, for example, some points in an extremity of a cluster might be closer to the points in the nearest cluster then to the points in the other extremity of the same cluster.

Implementation Details

Data Loading

In our implementation, we started by loading the data file using *pandas* and extracting both the latitude and longitude values from the file. Then we used the function provided to us in the project guidelines to transform our latitude and longitude values into x, y and z values (we transformed our 2d data into 3d data, so we could calculate the distance between the seismic events), using 6371 as our *RADIUS* value. Finally, we inserted those x, y and z values into our *data3d* matrix, each column corresponding to each variable.

K-Means Algorithm

The first algorithm we implemented was the *K-Means* algorithm, by simply calling the *KMeans* algorithm (implemented in the *Scikit-Learn* library) for each desired value of *k* (number of clusters) to fit our *data3d* matrix (**kmeans = KMeans(n_clusters=clusters).fit(data3d)**).

We also called the function *predict,* given our matrix, to get the labels of our data and the *silhouette_score* function, given our matrix and labels, to get the correspondent silhouette score value for the chosen *k* value.

Furthermore, we used the provided *plot_classes_mw* function to plot our data.

DBSCAN Algorithm

Given the more complex nature of this implementation we will provide a more detailed explanation of our procedure and show some key lines of code.

knn = KNeighborsClassifier(n_neighbors=K).fit(data3d, distances)

distances, _ = knn.kneighbors()

distances = distances.max(1)

In this segment, we call the KNeighborsClassifier for a given number of neighbors K to fit our data3d matrix and a previously created distances array. The distances array will then be filled by the values returned by the kneighbors function, which calculates the distances of each point to all its nearest K neighbors, becoming a matrix with K distances per row (each row corresponds to a point). Finally, we select only the max distance per row and obtain a distance array where we have this max distance for each point (this max distance corresponds to the distance each point has to its furthest K nearest neighbor point).

After this, we sort and scale the distances array (now it contains only values between 0 and 1).

At this point we need to implement the method to select the best elbow parameter. For starters, we need to calculate the derivate of our distance values, but in order to do so, we must "smoothen" our values first. Be smoothing our values, we mean to select more separated values from our array (values with indexes separated by ten is what we chose, as advised by the professor), as without smoothing, our derivatives would variate too radically.

So if we consider:

$$dY = Yi - Yi\text{-}1$$

$$dX = Xi - Xi\text{-}1 = 1/N \text{ (N - data points)}$$

Then (given Smoothing = 10):

dY = np.diff(distances[::Smoothing])

dX = 1/float(len(distances[::Smoothing]))

derivative = dY/dX

Now we define the method get_elbow_index, which give the derivatives array will return the first index of the array where the value of the derivate is higher than a threshold value (in our case we chosen the value 1: Param = 1).

```
def get_elbow_index(derivative):

for i in range(len(derivative)):

if(derivative[i] > Param):

return i
```

This returned index will correspond to the index of the distances array that will contain the value of our elbow. But first we need to multiply the index with the Smoothing value as the derivatives array only contains values ten in ten from the distances array.

elbow = distances[get_elbow_index(derivative) * Smoothing]

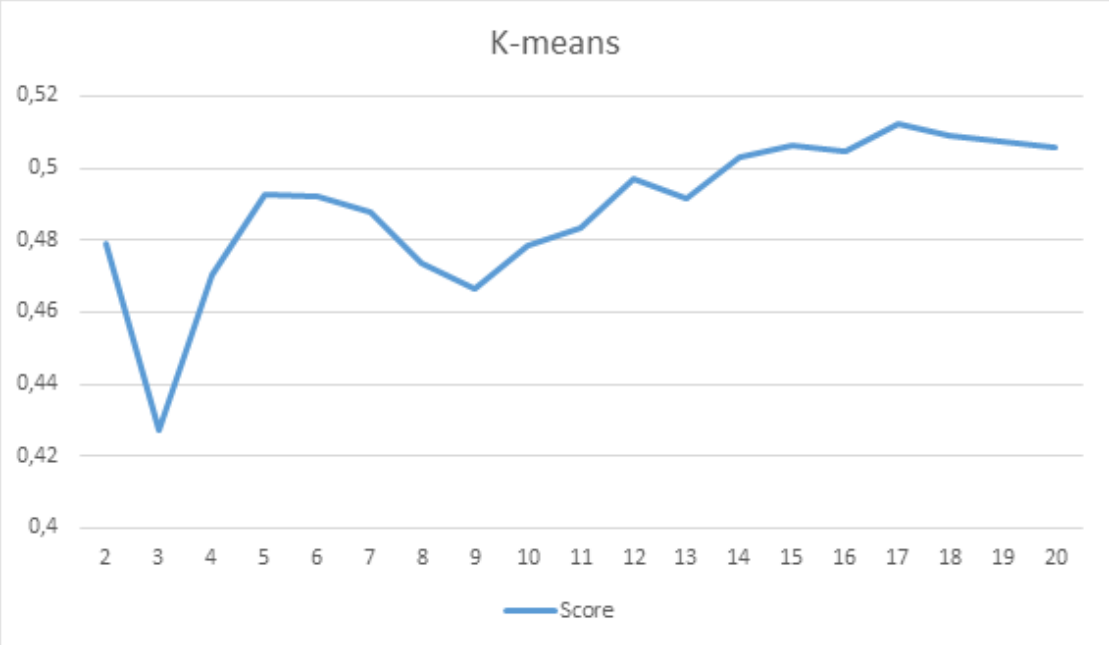
Now we just need to rescale our elbow parameter and we get the intended Eps value (maximum radius of the neighborhood):

eps = elbow * max_dist

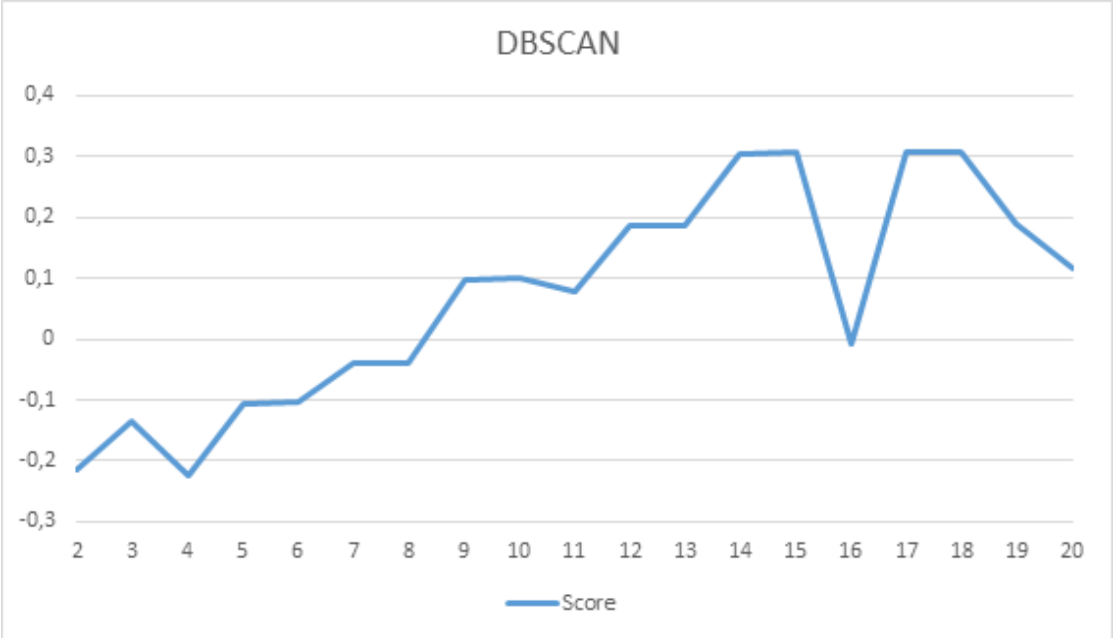
Finally, just like we did for the K-Means algorithm (now calling the DBSCAN algorithm, implemented in the *Scikit-Learn* library to fit our data), we calculate our labels and silhouette score. To plot our data, we also use the plot_classes_mw function.

Obtained Results

For the **K-Means** algorithm, we tested our program with the values of *k* ranging from 2 to 20. The values we obtained are shown in the graph below, where the y values represent the silhouette score and the x values the *k* value (representing the number of clusters):



For the **DBSCAN** algorithm, we tested our program with the values of *k* ranging from 2 to 20. The values we obtained are shown in the graph below, where the y values represent the silhouette score and the x values the *k* value (representing the value of *MinPts*):



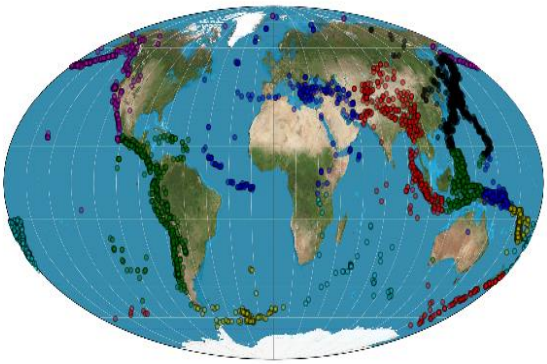
As we can see the silhouette score for both of these values was highest when *k* was 17. The **K-Means** algorithm obtained an overall better silhouette score than the **DBSCAN** algorithm, but as we said before, this score is not very adequate for this data set, as the clusters are not globular at all.

For both of the algorithms we tried other values of *k* before making the graphs, but we observed that they wouldn't get much better after 20, so we decided it wasn't necessary to include to get the general view of the results.

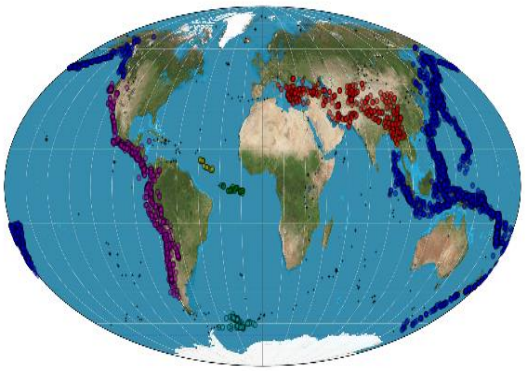
Conclusion

Even though the **K-Means** algorithm had a better silhouette score, we feel that the best algorithm to solve this problem is the **DBSCAN** algorithm, as the clusters it created made more sense in the context of the problem.

We can see below the comparing of the plot of the clusters for both algorithms, using the best *k* value (k = 17):



K-Means



DBSCAN