
# **I**ntroduction to **M**achine **L**earning

Activity 1

*Course 2020-2021*
*November 1st, 2020*

---

**Students:**

| | | |
|---|---|---|
| *Roger Marrugat* | - | *roger.marrugat@estudiantat.upc.edu* |
| *Victor Badenas* | - | *victor.badenas.crespo@estudiantat.upc.edu* |
| *Ronald Rivera* | - | *ronald.rivera@estudiantat.upc.edu* |

# Contents

# Overview

The document describes the implementation of unsupervised learning through the use of Clustering techniques and evaluates its accuracy. The clustering techniques utilized during the implementation of this project are DBScan, Fuzzy C-Means, K-means and modifications of K-means Bisecting and K-means++. To evaluate these techniques we use the datasets of adult (mix, large dataset), vote(categorical, small dataset), Pen-based(numerical, large dataset). To analyze the accuracy and optimal number of clusters we use the 4 different clustering validation metrics and Silhouette method, respectably.

# Execution Instructions

the configs folder contains the json required to run the main.py script. The json has the following structure:

```
{
    "path": "<relative_path_to_arff_file>",
    "resultsDir": "path_to_results_folder",
    "verbose": <bool>,
    "plotClusters": <bool>,
    "parameters": <nested_objects_with_execution_parameters>
}
```

With that JSON, the `main.py` file can be called using the following command:

```
python main.py -c <path_to_json>
```

This main script will run the algorithms for which the json object has parameters for and it will store the confusion matrices and the metrics extracted for each one of the algorithms in the `resultsDir` directory declared in the json file. It will also show and return the values of the labels as well as show how much time did the algorithms take to run.

Additionally, some files can be found in the scripts folder:

- 3dplotAndMetrics.py : script to generate PCA representations of the data with color coding for the clusters assigned by each algorithm
- datasetPlots.py : script to generate boxplots and scatterplots of the databases
- dbscanScript.py : script to find the bes optimal values for the DBScan parameters
- executionTimePlot.py : script to perform plots and metrics over the average runtime of each algorithm

# Reading Arff File

Data is stored in the arff format, which is usually used in WEKA. The format consists of a header containing information about the type of data present in the dataset and the data itself.

All the methods and attributes useful to load and modify the data in each arff file are encapsulated in the *ArffFile* class in the *src/dataset.py* python file. The object will also switch between the possible modifications that can be done to the data.

With the objective of reading the arff file into a pandas Dataframe for easier manipulation of the data inside python, we use the function in *scipy.io.arff* called *loadarff.* Once the data is correctly loaded into the *self.data* attribute of the class, the data is formatted according to the parameters given to the ArffFile Object in the constructor. Those values are *stringConversion* and *floatNormalization.* StringConversion specifies the methodology used to perform the conversion of the categorical features to numerical. The floatNormalization parameter specifies the methodology for converting integer features to floating point values respectively.

For the *stringConversion* parameter there are two supported methodologies:

- *int*: converts categorical string features to integer values in the range (0, num_classes-1).
- *onehot*: converts categorical string features to vectors of num_classes positions with a 1 in the position belonging to each class and 0 elsewhere.

Observing the datasets, unknown values in the categorical data are represented by a character which before performing either of the previous modifications to the feature data, has to be replaced with some meaningful value. In this case, the object replaces the value with the most common value.

For the floatNormalization parameter there are four supported methodologies:

- *standardisation*: This option will scale to normalise by the mean and variance by subtracting the mean and dividing by the standard deviation.
- *mean*: Same computation than standardization using another package. Only added for completeness.
- *min-max*: Normalizes data linearly to be between 0 and 1 by adding the min of the data and dividing by the range of the data.
- *unit*: normalized by dividing by the L2 norm.

From this, once all the data is normalized between 0 and 1 as float values, the formatted data can be extracted using the *getData()* method.

# DBSCAN

DBScan is a density-based clustering algorithm that tries to create partitions from dense regions of data points that are separated by lower density areas. Contrary to K-means, it doesn't need clusters to be convex shaped and it handles much better noise or outliers, but it still needs user-defined parameters in order to find the optimal degree of density areas necessary to form a cluster.

In DBScan we define 3 types of points and 2 parameters:

- A core point has more neighbour points (distance lower than $\varepsilon$ ) than the specified **MinPts**.
- A border point doesn't reach the necessary number of neighbour points but is the neighbour of a core point.
- A noise point neither reaches **MinPts** nor is the neighbor of a core point.

The idea is to find the optimal values for $\varepsilon$ and **MinPts** that maximize the density of the clusters, while minimizing the density of the middle areas.

- If the $\varepsilon$ is small and **MinPts** is large, the vast majority of points will be considered outliers.
- If the $\varepsilon$ is small and **MinPts** is small too, generated clusters will only contain core points that are really close to each other, and there will be lots of them.
- If the $\varepsilon$ is large and **MinPts** is small, lots of points are considered core points and, if they are close enough, they all belong to the same cluster.
- If the $\varepsilon$ is large and **MinPts** is large too, all the points will be included in a single huge cluster.

In order to determine these optimal values we need to calculate the distance to the nearest neighbor K for each point[1]. We do this under the hypothesis that distances for points in a cluster to their Kth nearest neighbor are the same, while noise points are much more distant to their Kth neighbor. To compute the distances, we can use the K Nearest Neighbors algorithm.

Once the distances are obtained, we sort them to find the point of maximum curvature that indicates the radius necessary to include the border points of the cluster. The following points in the curvature represent noise points. At this point of the plot, the $\varepsilon$ is the Kth Nearest Neighbor distance while **MinPts** is the parameter K.

[1] https://iopscience.iop/article/10.1088/1755-1315/31/1/012012/pdf

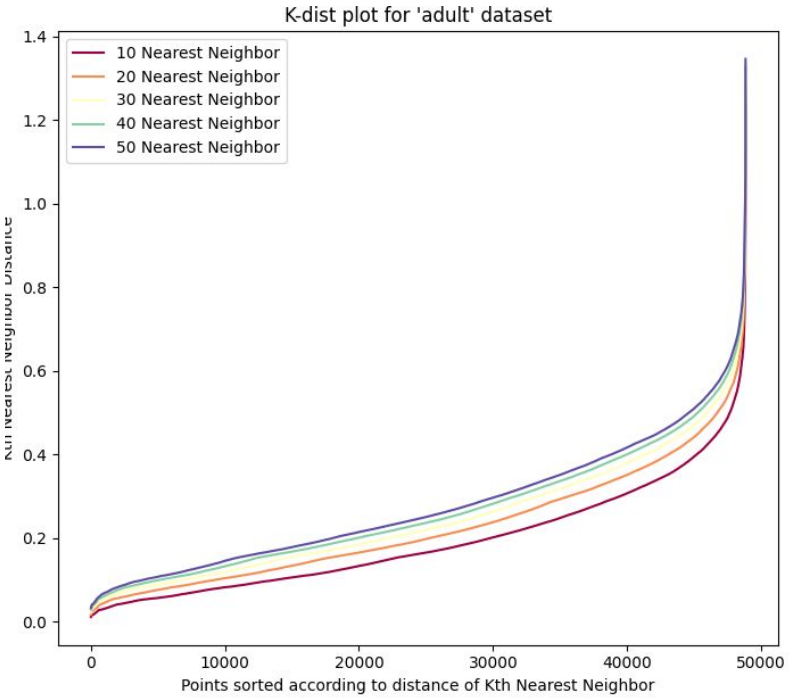


Figure 1: K-dist plot for Adult Dataset

| | -1 | 0 | 1 |
|---|---|---|---|
| -1 | 0 | 0 | 0 |
| 0 | 2 | 22732 | 14421 |
| 1 | 42 | 9912 | 1733 |

Table 1: Dataset: adult.arff (0.75 $\varepsilon$ , 20 MinPts - y = Goal Standard, x = DBScan clusters)

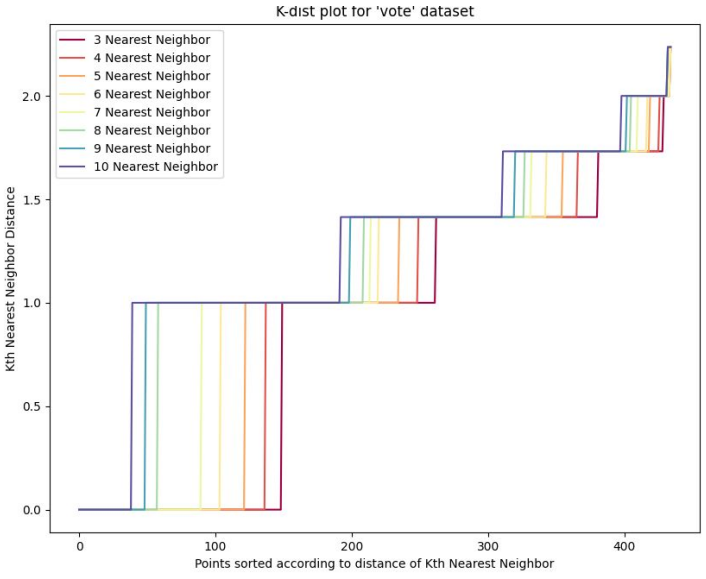


Figure 2: K-dist plot for Vote Dataset

| | -1 | 0 | 1 |
|---|---|---|---|
| -1 | 0 | 0 | 0 |
| 0 | 29 | 31 | 207 |
| 1 | 14 | 147 | 7 |

Table 2: Dataset: vote.arff (1.5 $\varepsilon$ , 8 MinPts - y = Goal Standard, x = DBScan clusters)

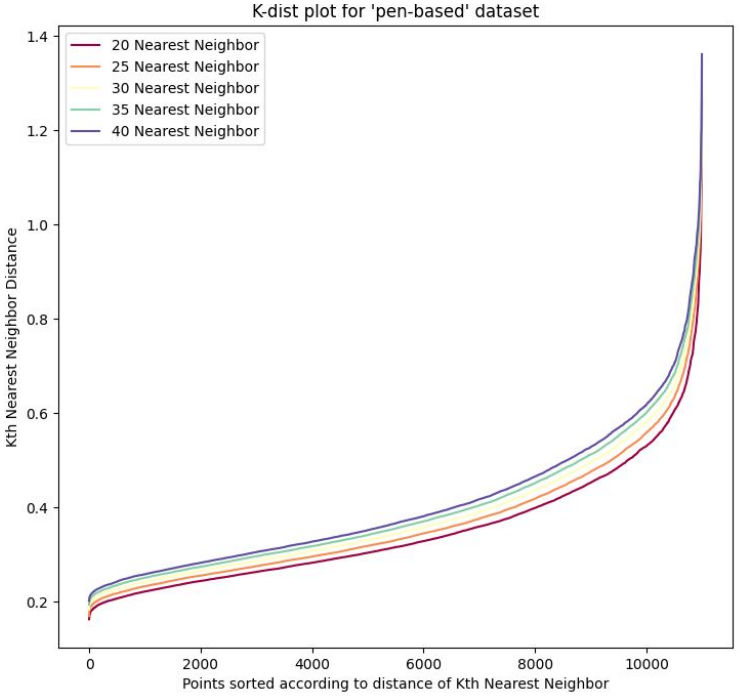


Figure 3: K-dist plot for Pen-Based Dataset

| | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 80 | 1062 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 123 | 0 | 658 | 280 | 82 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 13 | 0 | 3 | 1128 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 8 | 0 | 1046 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 79 | 0 | 0 | 0 | 0 | 1063 | 2 | 0 | 0 | 0 | 0 | 0 |
| 5 | 47 | 0 | 236 | 0 | 0 | 0 | 170 | 602 | 0 | 0 | 0 | 0 |
| 6 | 29 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1025 | 0 | 0 | 0 |
| 7 | 103 | 0 | 136 | 903 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 772 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 223 | 37 | 22 |
| 9 | 562 | 0 | 65 | 0 | 0 | 1 | 427 | 0 | 0 | 0 | 0 | 0 |

Table 3: Dataset: Pen-Based.arff (0.415 $\varepsilon$ , 35 MinPts - y = Goal Standard, x = DBScan clusters)

# Algorithms

## K-Means

The objective of the k-means algorithm is to separate the data given to k clusters. Each cluster is defined by a point in the feature that represents the center of the cluster. Each data point is assigned to the cluster whose center is closer to the data point as measured by a distance metric.

The main goal of the algorithm is to minimize the distance between the data points inside each cluster. To do that, multiple passes on the data are done. In each of the epochs, the centers are recomputed to minimize the function. The algorithm looks as follows:

1. initialize the centers
2. select clusters
3. recompute centers
4. check for convergence, if not, return to 2

The steps define 4 functions: the center initializer, the cluster selector, the center computer and the convergence checker functions.

The cluster initializer is the method responsible for commuting between the possible methods of initialization. The options for the k-means algorithm are:

- *random*: will initialize the k centers needed with k random data points of the training data.
- *first*: will initialize the k centers with the first k data points in the training data
- *K-Means++*: explained in its dedicated section.

The method to select clusters assigns a cluster index (which corresponds to the index of the center's cluster in the centers array) to each data point in the data. This method is used by both the *fit* and *predict* methods. This assignment is done by computing the distance between all the data points and all the clusters and then selecting the index with the minimal distance for each data point. In this case, the distance used is the L2 or euclidean distance.

The function responsible to recompute centers is called each iteration in order to update the centers to a vector which is better than the previous ones. This is done by taking all data points that have been assigned to each cluster and computing the mean vector between them.

The convergence checker function will return *True* if the algorithm must exit and *False* if a minima in the cost function has not been reached and the algorithm must continue. The function must return True if one of the following criteria are met:

- The centers have not changed from one iteration to another or have changed less than a tolerance value defined in the constructor.
- The data points are assigned the same cluster indexes in two consecutive iterations.

An addition was done to the algorithm after checking that the performance of the algorithm was not very consistent. An additional parameter called *nInit* was defined which defined how many times the k-means algorithm was to be run before selecting the best run from them and returning it as the result of the *fit* function. This proposed the issue of defining a metric to evaluate the performance to be able to select which is the model with the best metric among the possible ones. The inertia value was the one to fulfill this purpose. The inertia value, as defined in[2], can be computed as the sum of all intra-cluster distances for each one of the clusters and then add the distances to obtain a numerical metric. This metric is stored in an array in each iteration and then the one with the best final inertia value (when the algorithm exits because it has converged) is the computation selected to be returned as the best model.

As it can be seen in the KMeans class docstring, the parameters that the algorithm takes in the constructor are:

- n_clusters (int, default=8): The number of clusters to form as well as the number of centroids to generate.
- max_iter (int, default=300): Maximum number of iterations of the k-means algorithm for a single run.
- init ({'random', 'first'}, default='random'): Method for initialization:
- n_init (int, default=10): Number of times the algorithm will run with different centroid seeds. The best result in terms of inertia will be preserved
- tol (float, default=1e-4): Maximum value tolerated to declare convergence by stability of the centers
- verbose (bool, default=False): Verbosity mode.

And the class main methods:

- fit(trainData): computes the centers n_init times and returns the best kmeans object in terms of inertia.
- predict(data): returns labels assigned to each data point in data.
- fitPredict(data): performs fit and returns the labels as in predict.

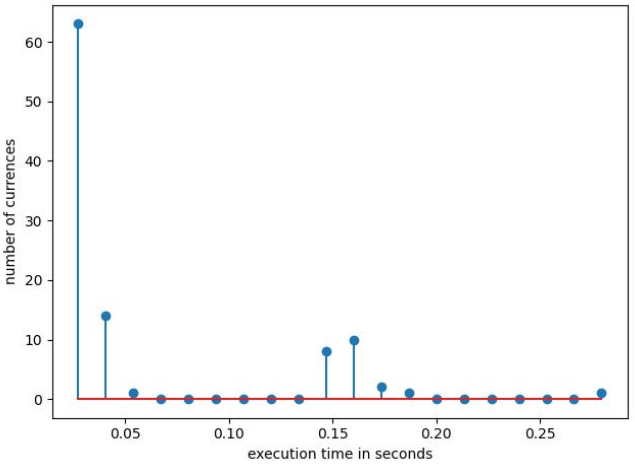


Figure 5: Execution time histogram for K-Means algorithm

[2] https://towardsdatascience.com/k-means-clustering-from-a-to-z-f6242a314e9a

The algorithm was run 100 times over the *vote.arff* dataset to measure statistics on the time performance of the algorithms. The mean runtime is 0.06s with a standard deviation of 0.06s for a clustering run with K=3 on the vote dataset.

## Bisecting K-Means

Bisecting K-Means is an example of divisive hierarchical clustering. The goal of the algorithm is to divide the dataset points into clusters of equal size. It does that by extending the K-Means clustering algorithm.

The algorithm uses K-Means to segment the clusters. As the name suggests, the algorithm builds the dendrogram of clusters by splitting the biggest cluster in two. To do this, KMeans with *n_clusters* set to 2 is used to separate the data in a cluster. The algorithm has the following steps:

1. initialize cluster
2. assign cluster indexes to data
3. compute biggest cluster and its data
4. compute KMeans in biggest cluster data
5. remove biggest cluster and add kmeans centers
6. if the number of clusters is smaller than k, return to 3

The initialize cluster method initializes all the data points to a cluster by only defining one cluster center. The center is initialized by assigning it to a random point in the input data. The cluster assignment is computed with the same methodology as KMeans. The biggest cluster computation is performed by computing the distance between the center of each cluster and the furthest point in it.

After that, an argmax function returns the index of the cluster with the furthest data point. The K-Means algorithm is reset and then computed. After that, the biggest center is popped from the centers array and the centers from the K-Means algorithm are appended to it. Finally, the previous methods are iteratively run until there are *n_clusters* centers in the array.

Parameters in the BisectingKMeans are mostly passthrough parameters for the KMeans algorithm to bisect the clusters. The only parameter that is exclusively for the bisecting K-Means is the number of clusters, which will determine the number of center arrays that are needed. The algorithm was run 100 times with 3 clusters and the vote dataset. The mean execution time 0.05s with a standard deviation of 0.05s.

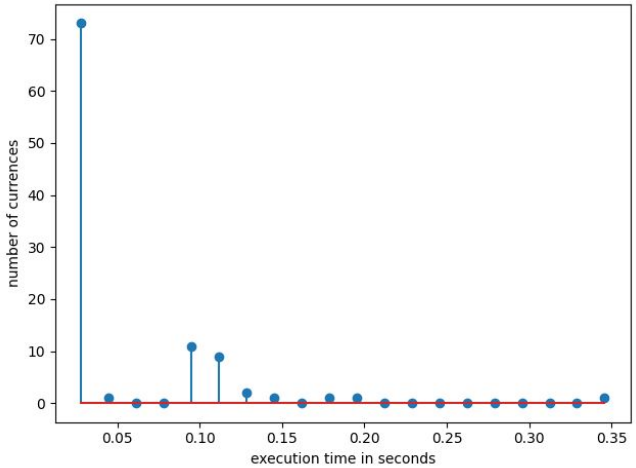


Figure 6: Execution time histogram for bisecting K-Means algorithm

## K-Means++

K-Means++ as we mentioned before is modification of the base K-means algorithm with a modified initialization. This modification is focussed on addressing a shortcoming that the original K-means have of the initialization sensitivity of the centroids.

The algorithm starts by selecting a random centroid in the provided data, after we have our first centroid we calculate the distances from all the points to the previous centroid. Then we select the point with the farthest distance to the previous centroid and is selected as the new centroid, then we repeat the previous steps until we get the number of centroids desired. After reaching the desired numbers of centroid we proceed to implement the standard K-means algorithm. The algorithm has the following steps:

1. Randomly select the first centroid
2. Compute the distance from the nearest point points of the previous centroid
3. Select the next centroid from the point that have the maximum distance from the previous centroid
4. Repeat steps 2 and 3 until k centers have been chosen
5. Proceed to implement original K-means method using the previous selected centers

Parameters in the KMeansPP are similar to our BisectingKMeans are passthrough parameters for the KMeans algorithm to add the initialization centroids. The number of clusters is the only parameter that is exclusively for K-Means++, which will determine the number of initials centroids would be calculated.

The K-means++ was also run 100 times over the same dataset and was executed in a mean of 0.03s with a standard deviation of 0.004s.

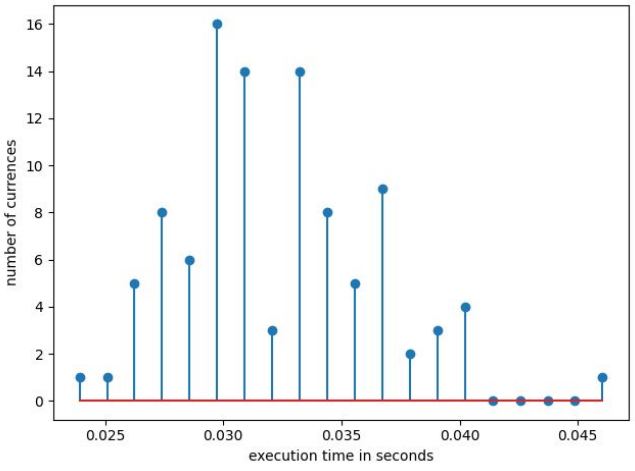


Figure 7: Execution time histogram for K-Means++ algorithm

## Fuzzy C Means

The FCM algorithm is an example of a fuzzy clustering technique where the data points are not assigned a cluster but instead they are assigned a membership value to each cluster. This way, the data point will be assigned a vector of *n_clusters* size containing the values for the membership of the data point to each cluster.

The algorithm runs with the next steps:

1. initialize Membership matrix
2. update cluster centers
3. compute new Membership matrix
4. check for convergence, if not return to step 2

This can be achieved by defining functions for each step:

1. The Membership matrix (U matrix) has to be initialized. The membership matrix is an array of shape (n_points, n_clusters) with a membership value in each cell for each data point and each cluster. This matrix has the restriction that the sum of the membership values for each data point must be equal to 1. This restriction has to be taken into account when generating the random matrix, as the random matrix has to be normalized by the sum of each axis to force a sum of 1.

2. The centers are updated following the following formula:

$$v_i = \frac{\sum_{k=0}^{n-1}(u_{ik})^m x_i}{\sum_{k=0}^{n-1}(u_{ik})^m}$$

   The mathematical expression expresses the center i as a function of the $u_i$ vector and the data point i. However the brute force implementation of the formula is very inefficient and for that, we vectorized the operation in order to be able to run it faster. In the vectorized form of the operation, the U matrix is first raised to the m power and afterwards, the sum of the denominator for each value of i is computed as the sum of the column values of the u matrix. After that, the numerator is calculated by using the dot product of the U matrix transposed by the data.

3. The new membership function is computed as the following function:

$$u_{ij} = \frac{1}{\sum_{k=1}^{C}(\frac{d_{ij}}{d_{ik}})^{\frac{2}{m-1}}}$$

   This is the most complex computation of the algorithm, as if the calculation of the values of the matrix is performed one by one it is really slow. For this matter, the computation is vectorized the following way: First, the $d_{ij}$ matrix is computed as the distance between all data points and all centers of the PCM algorithm. Then, because we need to compute the distance ratio between all values of i,j and k, the matrix has to be replicated k times over a new axis in order to have the same data repeated k times in the denominator. Once the data is created and replicated, we can see that the first axis in the numpy array corresponds to the i index in the equation, the second axis corresponds to the j index in the equation and the last one corresponds to the k index. Because of that, the dik matrix is replicated in the second axis, as it

has to be invariant in the j value. The denRatio for each i, j and k is then computed by dividing the dij (expanding the last dimension for the element wise division to work in numpy) matrix by the dik matrix and then raised to the desired power. After the power operation, the sum in the second axis (which belongs to k) is done and the matrix is inverted elementwise to finally return the U membership matrix.

```
dij = cdist(data, self.centers)
dik = np.repeat(dij[:, np.newaxis, :], self.nClusters, axis=1)
denRatio = dij[:, :, np.newaxis] / dik
denRatio = denRatio  ** (2/(self.m-1))
return 1 / np.sum(denRatio, axis=2)
```

4. Finally for the check for convergence step, the norm is computed between the last step and the current step membership matrixes. if that norm is smaller than a tolerance value, the algorithm stops iterating and returns the centers.

The fuzzyCMeans algorithm was also run 100 times with C=3 and the time taken to run was 0.022s in mean with a standard deviation of 0.004s.

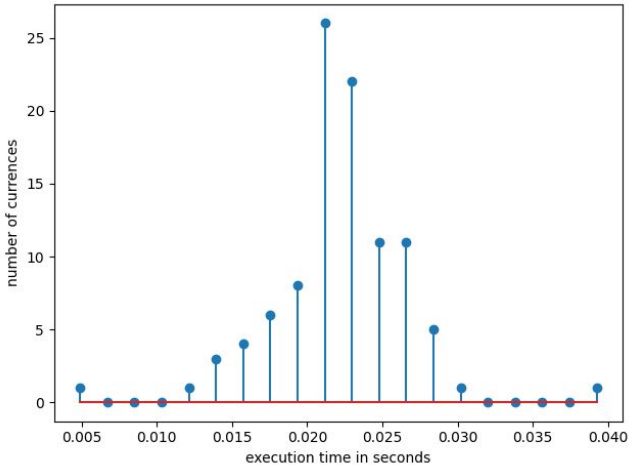


Figure 8: Execution time histogram for fuzzy C-Means algorithm

# Cluster Mapping Metric

Apart from all the metrics that will be used to evaluate the performance of the algorithms, we propose a method called cluster mapping metric, which is an attempt of regularizing the number of clusters to the ground truth clusters present in the datasets to perform a supervised metric of the performance of the algorithm.

The goal of the cluster mapping metric is to create a square confusion matrix for all labels in the data points. This was created to overcome the issue of having different indexes in the ground truth with relation to the indexes in the predicted labels by the algorithm.

In order to regularize the clusters on the predicted cluster indexes, we propose to reassign a cluster index from the ground truth to each of the clusters in the predicted labels array. The criteria according to which the mapping is made is the most common ground truth label of the data points in the cluster will be the cluster index reassigned to that cluster.

```
for clusterIdx in set(labels_pred):
   mask = labels_pred == clusterIdx
   clusterDataGs = labels_gs[mask]
   clusterAssignation = Counter(clusterDataGs).most_common(1)[0][0]
   modified_labels[mask] = clusterAssignation
return confusion_matrix(labels_gs, modified_labels)
```

code responsible for the index mapping

After the cluster indexes are reassigned to fit the ground truth number of classes, the confusion matrix can be computed and returned successfully, yielding a nxn confusion matrix.

# Results Obtained

In the next section we would discuss the results obtained for each technique used. For determining the optimal K value we implement the Silhouette method. The Silhouette method is used to measure how well a data point fits a cluster based on the closeness of the data point to other data points in the cluster and how far the data point is from other points in other clusters. Larger values represent sample data that is closer to its cluster than other clusters. We the Silhouette coefficient obtained in our calculations the optimal number of clusters for each clustering technique and dataset is showcased in Table 4 below.

| Optimal K Value | Adult | Vote | Pen-Based |
|---|---|---|---|
| KMeans | 9 | 7 | 8 |
| KMeans++ | 8 | 6 | 10 |
| Bisecting KMeans | 5 | 7 | 6 |
| Fuzzy C Means | 8 | 6 | 6 |

Table 4: Optimal value of K (number of clusters) for each dataset

To evaluate each method we used the following clustering validation metrics:

- ***Davies Bouldin Score***, average simiriality value for each cluster with its most similar cluster. **Lower is better for this metric.**
- ***Adjusted Rand Score***, similarity between two clusters by considering sample and counting pairs from the same cluster or a different one. **Higher is better for this metric.**
- ***Completeness Score***, score based on if all data of the same class are in the same cluster. **Higher is better for this metric.**
- ***Purity Score***, each cluster is assigned a class based on what is the most frequent class of the data points inside the cluster, we calculate the purity by evaluating how many point classes are assigned correctly in the cluster. **Higher is better for this metric.**

In the below Table 5 and 6 is present the clustering validation metrics results for each clustering method using the optimal number of clusters from the previous session.

| Clustering Method | Metric | Adult | Vote | Pen-Based |
|---|---|---|---|---|
| DBScan | davies_bouldin_score | 1.482 | 2.349 | 1.645 |
| | adjusted_rand_score | 0.031 | **0.518** | **0.533** |
| | completeness_score | 0.042 | **0.365** | **0.748** |
| | purity_score | 0.762 | 0.880 | 0.681 |

Table 5: Clustering Validation Metrics for DBScan for each dataset

| Clustering Method | Metric | Adult | Vote | Pen-Based |
|---|---|---|---|---|
| K-Means | davies_bouldin_score | **1.805** | 2.192 | 1.291 |
| | adjusted_rand_score | 0.018 | 0.200 | 0.515 |
| | completeness_score | 0.047 | 0.230 | 0.697 |
| | purity_score | **0.768** | 0.920 | 0.655 |
| K-Means++ | davies_bouldin_score | 1.814 | 2.147 | **1.269** |
| | adjusted_rand_score | 0.035 | 0.219 | 0.532 |
| | completeness_score | **0.050** | 0.222 | 0.699 |
| | purity_score | **0.768** | 0.913 | **0.707** |
| Bisecting K-Means | davies_bouldin_score | 1.940 | **2.060** | 1.640 |
| | adjusted_rand_score | -0.020 | 0.410 | 0.260 |
| | completeness_score | 0.030 | 0.270 | 0.550 |
| | purity_score | 0.760 | **0.920** | 0.400 |
| Fuzzy C-Means | davies_bouldin_score | 3.320 | 2.669 | 3.801 |
| | adjusted_rand_score | **0.053** | 0.274 | 0.115 |
| | completeness_score | 0.030 | 0.255 | 0.451 |
| | purity_score | 0.766 | 0.880 | 0.239 |

Table 6: Clustering Validation Metrics for optimal K (number of clusters) for each dataset

Based on the validation metrics obtained we classify the best clustering method for each dataset as follows:

- **Adult Dataset Results:** With the results obtained from the alone metrics we could not establish which clustering method is better due to having satisfactory values in all for methods. For this reason we have used the confusion matrices (Tables 7-8) for each clustering method presented below. After calculating the accuracy (Table 9) of each matrix we can conclude that **K-Means++ with 8 clusters** for this dataset.

| K-Means (k=9) | 0 | 1 |
|---|---|---|
| **0** | 35283 | 1872 |
| **1** | 9437 | 2250 |

Table 7: Confusion Matrix of Adult Dataset using K-Means (k=9)

| K-Means++ (k=8) | 0 | 1 |
|---|---|---|
| **0** | 35281 | 1874 |
| **1** | 9438 | 2249 |

Table 8: Confusion Matrix of Adult Dataset using K-Means++ (k=8)

| Adult Clustering Method | Accuracy |
|---|---|
| **K-Means (k=9)** | 0.768 |
| **K-Means++ (k=8)** | 0.768 |

Table 9: Accuracy values for each Clustering Methods in Adult Dataset

- **Vote Dataset Results:** Based on the results obtained in Tables 5 and 6, we can presume that **DBScan** would provide the best clustering. After confirming with the Confusion Matrix (Table 10) and its accuracy(Table 11) of 88%, confirm our supposition.

| DBScan | 0 | 1 |
|---|---|---|
| **0** | 236 | 31 |
| **1** | 21 | 147 |

Table 10: Confusion Matrix of Vote Dataset using DBScan

| Vote Clustering Method | Accuracy |
|---|---|
| **DBScan** | 0.88 |

Table 11: Accuracy value for DBScan in Vote Dataset

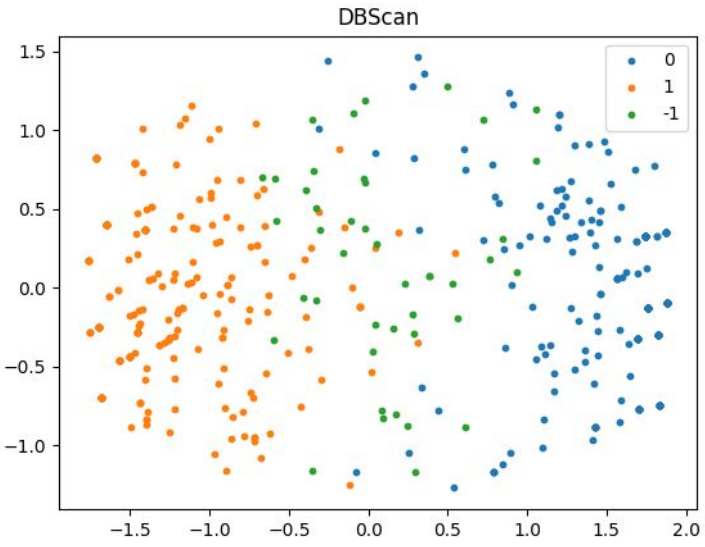


Figure 9: Scatter Plot with PCA for DBScan in Vote Dataset

- **Pen-Based Dataset Results:**Using the result in Table 5 and 6, we have a strong indication that the best clustering method is **K-Means++ with 10 clusters**. Incorporating the Confusion Matrix (Table 12) and its accuracy confirm our assumption with 70% accuracy (Table 13).

| KMeans++ (k=10) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 711 | 2 | 1 | 0 | 7 | 0 | 14 | 0 | 406 | 2 |
| **1** | 0 | 643 | 333 | 87 | 1 | 0 | 9 | 0 | 0 | 70 |
| **2** | 0 | 16 | 1128 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **3** | 0 | 24 | 1 | 1027 | 1 | 0 | 0 | 0 | 0 | 2 |
| **4** | 0 | 13 | 1 | 1 | 1046 | 0 | 51 | 0 | 0 | 32 |
| **5** | 0 | 0 | 0 | 235 | 0 | 624 | 6 | 0 | 3 | 187 |
| **6** | 0 | 0 | 0 | 0 | 3 | 1 | 1052 | 0 | 0 | 0 |
| **7** | 0 | 158 | 900 | 77 | 1 | 4 | 1 | 0 | 1 | 0 |
| **8** | 16 | 0 | 32 | 106 | 0 | 22 | 6 | 0 | 869 | 4 |
| **9** | 11 | 82 | 9 | 199 | 88 | 0 | 1 | 0 | 1 | 664 |

Table 12: Confusion Matrix of Pen-Based Dataset using K-Means++ (k=10)

| Pen-Based Clustering Method | Accuracy |
|---|---|
| **K-Means++ (k=10)** | 0.706 |

Table 13: Accuracy value for K-Means++ (k=10) in Pen-Based Dataset

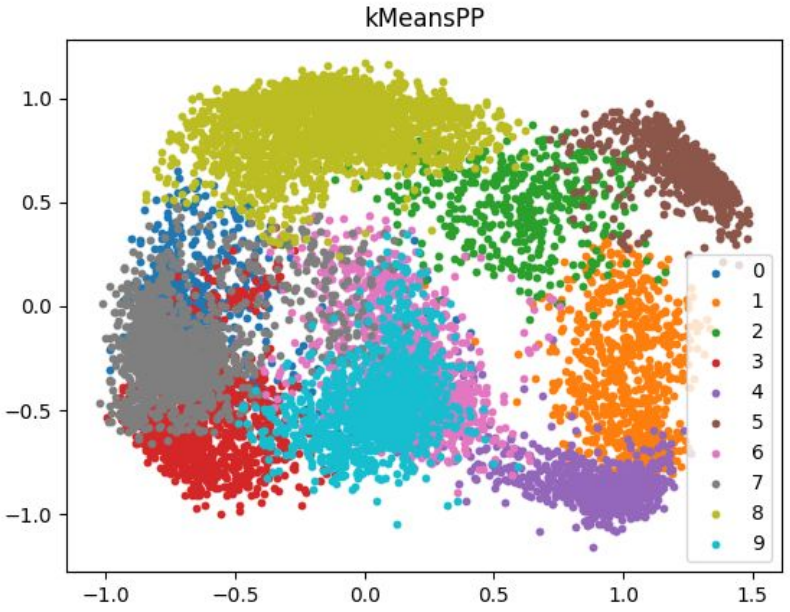


Figure 10: Scatter Plot with PCA for K-means++ in Pen-Based Dataset

# Results Analysis

After evaluating the results obtained we analyzed information obtained for each dataset using each clustering algorithm.

- DBScan:
    - Adult: There's 2 big clusters, where one is at least two times as big as the other. These clusters are dense and their points are close to each other as the $\varepsilon$ =0.75 and MinPts=20. Not many outliers.

    - Vote: There's 2 clusters that can be identified quite nicely. There's clear and separated areas of points (as the k-distance plot shows) where each point has a big set of points really close, another big set of points at distance 1.0, at 1.5, at 2.0 and finally some outliers. With a big cluster radius $\varepsilon$ =1.5 and few neighbor points MinPts=8, the algorithm makes an effective separation (as the slope of distance 1.5 represents the border points). Not many outliers.

    - Pen-based: There's 10 clusters that can be identified correctly, except for the last 2 that contain not-so-dense regions (and result in outliers) and some clusters that get mixed with each other. Clusters are quite dense and their points are close to each other as the $\varepsilon$ =0.415 and MinPts=35. Many outliers in the last 2 clusters (points that are quite far).

- K-Means, K-Means ++ and Bisecting K-Means:

    - Adult: From the results that we obtained for the adult dataset, we can know that the two classes in this dataset are not well separated in two distinct and easily separable clusters. The fact that the algorithm did assign most of the data points to the same cluster for small k values leads us to believe that there are multiple clusters in the feature space that are formed by small groups of data points.

    - Vote: The good performance of the KMeans algorithm and its derivations on the vote dataset leads us to believe that there are two very distinct clusters in the data with some smaller clusters that are more ambiguous. The KMeans algorithms achieved over a 92% of accuracy on the training data, which is quite good.

    - Pen-Based: There are some clusters that can be identified correctly but it seems like there are some data points outside of the main clusters that are hard to clusterize and end up often misidentified as belonging to another cluster.

- Fuzzy C-Means:
    - As the Fuzzy CMeans has not achieved better results than the k-Means algorithms, we cannot infer any more information from these datasets.

In addition, we have evaluated the performance of each clustering algorithm based on the type of data contained in the datasets. We based our conclusion on the observed results, and theoretical analysis of each algorithm. Based on our result obtained we can conclude the following:

- **Categorical data (Vote Dataset):** we observed that DBScan was the best clustering method with a high accuracy above 88%.
- **Numerical Data (Pen-Based Dataset):** it was observed that K-Means++ have the best accuracy and satisfactory clustering validation metrics.
- **Mixed Data (Adult Dataset):** For mixed data it is observed that all clustering methods work well with high accuracy but the best method is K-Means++ with 76% accuracy.

From a theoretical standpoint the weakness algorithm is standard K-means because all of the other iterations are improved versions of it. The reason why categorical data does not work well with the standard K-means algorithm is because categorical data follow a discrete behavior, which a distance based algorithm would not work well.

For mixed data the best algorithm would be based on the distribution of numerical and categorical data present in the dataset. With this assumption it is recommended to utilize algorithms that use the mode of the datasets to evaluate categorical data. A mixed dataset with a high amount of categorical points would be better a categorical focus algorithm and vice-versa.

Finally, we evaluate if there is any difference among the algorithms based on the dataset chosen. All algorithms are used to group the data points into groups of certain similarity following a set of mathematical rules. However, those mathematical rules make it so that the algorithms behave completely differently to the same data.

For Instance, K-Means usually clusterizes the data by computing the minimum distance to the centers (partitioning based) while DBScan is an algorithm that attempts to find areas of high density of points surrounded by regions where the density is very small (density-based). Both algorithms will work quite differently and will fit a different data profile.

Each algorithm can provide some sort of underlying information of each dataset. For example, DBScan describes the density of the data points as well as the number of outliers or noise for each class.

However, when the points are not so densely distributed and the dataset contains classes with points separated between them, partition-based methods as K-means, K-means++ or

Bisecting K-means may render more effective than DBScan as their centroids may represent a better generalization of the points of the cluster or the class.

If the data points of each class may be mixed with each other in the spatial distribution, which makes it harder for Hard Clustering methods as the points may be at the same distance from two different datasets. In those cases, Soft Clustering Methods as Fuzzy C-Means may provide a better understanding of the underlying data as the membership functions of a point to several clusters will provide similar values.

Concerning the Adult dataset, none of the Clustering methods is able to be extremely effective given the particularity of the dataset, as there are 2 different classes whose points are mixed with each other. Even though there's 2 big dense and separated areas which DBScan is perfectly able to discern, each one of those areas contains points from both classes, rendering the results ineffective. K-means and K-means++ are able to find different clusters within those dense regions, increasing the performance of those algorithms over the dataset.

As for the Vote dataset, the underlying distribution of the dataset (republican vs democrats) suggests that there's 2 dense and clear regions (core voters for each party) and some outliers or points in the middle (indecisive voters and so on). A DBScan Clustering with optimal parameters is able to identify those 2 regions and classify the middle ones as outliers, as including those middle points in one of the clusters could mean that one cluster starts overlapping over the data points of the other. K-means++, Bisecting K-means or Fuzzy C-means are also able to correctly classify the 2 dense regions while gathering the middle points or outliers in their own clusters.

The pen-based dataset represents a data set of labeled handwritten digits where there are some areas with high density of points which are the most representative of a certain digit. As they are handwritten, they are close to each other and there are a lot of noisy points in the data. This is due to the similarity of the digit's shapes. The DBScan is quite ineffective (the eps value has to be set to a really small value in order not to have only one big cluster) as there are no low density areas and all the points are close to each other. The K-Means algorithm and its derivatives work quite well because if they achieve to locate a centroid inside each density region, they will be able to separate the zones as they don't take the density of the area into account to compute the clusters. The Fuzzy C Means algorithm is not very intuitive to use in this problem, as the clusters seem quite crisp in theory. A data point can either be a certain number or not. There is no degree of membership in the decision.