**ASSIGNMENT 5**

1) Given a large number of files containing positive integers we need a single MapReduce process to compute the percentage of even integers across all files.

Here I have solved the problem by just one mapper and reducer. We can use multiple mapper to increase the efficiency.

Mapper:

The First mapper takes the fileID and content as the input. It calculates the total number of even numbers and also the total numbers present for each file.

Input: key = fileID, value = content

Output: key = 1 , value = < total_even_no, total_no>

Pseudo Code for mapper

```
Input< fileID, file content>
        total_even = 0
        For ( Integer value: file content){
                If( value % 2) {
                        total_even ++
                }
        }
        Emit(1,< total_even, file.length>);
```

Reducer:

It reducer uses the input emitted from the mapper as the input which is key = 1 and value = < total_even, file.length>

The value is an array of pairs of even number count in each file and file.length which is the total number of integers present in that particular file. These pairs are summed up to get the total even numbers as well as overall total in all the files.

Input: key = 1 , value = < total_even_no, file.length >

Output: key =1 value = % of even numbers in all the files.

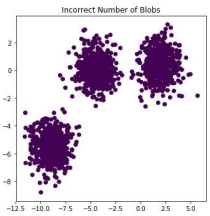
Pseudo Code for reducer

```
Input< 1, < total_even_no, total_no>[] value>
        Final_total_even = 0
        Final_total_no = 0
        For ( Integer n: value){
                Final_total_even + = n.total_even_no
                Final_total_no + = n.file.length
        }
        Emit(1, (Final_total_even/ Final_total_no)*100);
```

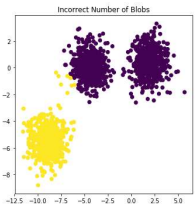
2) clustering

a) The example contains 4 code snippets running of different data points. We will change the number of clusters for each of the snippets and look at what's happening.

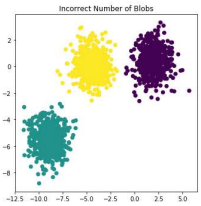
K-mean



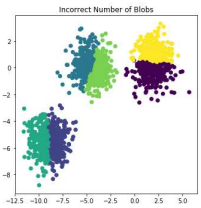

K=1



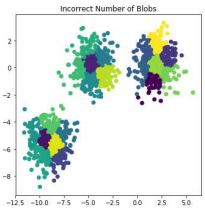

k=2




k=3

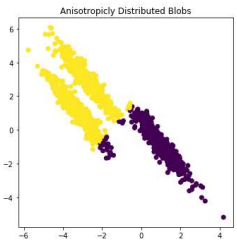



k=6

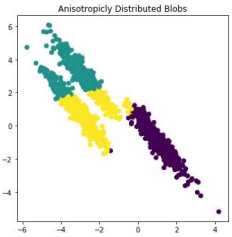



k=20

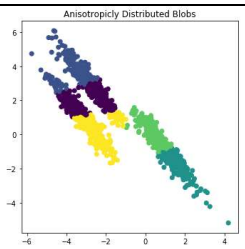
| | |
|---|---|
| k = 1 | all the data points belong to the same single cluster. Here only 1 centroid is chosen hence all the points should basically belong to the same cluster. This is never useful in practical situation. Just explains how number of centroid determine how many clusters we will have in the data. |
| k = 2 | the data points are clustered into 2 groups. When choosing the value, I thought the middle cluster would be split into 2 but we can notice that only few yellow data points clustered with the purple group This is because these data points are closer to centroid of purple cluster compared to that of the yellow cluster. |
| k = 3 | we can visualize 3 clusters. The data points are grouped compactly and works as expected. The data points are normally distributed and isotropic making it a perfect sample for k-means to cluster. (best solution) |
| k = 6 | we can notice that each of the 3 groups gets split off into two making 6 clusters. The algorithm calculates the average of all the points in a cluster and moves the centroid to that average location thus splitting the data points. |
| k= 20 | we have too many clusters when we increase the number of centroids and it's hard to think how these clusters would be split. The points are grouped making the data more granular. The cluster formation here is basically depends on the initial cluster centers. One other point that can be noted here is that the cluster do not form in between the data. |



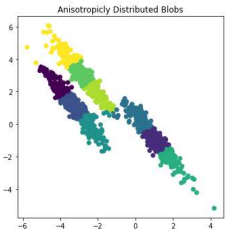

K=2



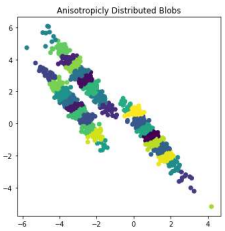

k=3




k=6

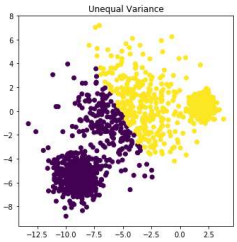



k=9

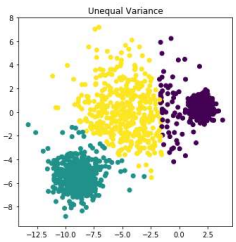



k=30

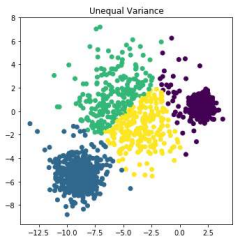
| | |
|---|---|
| k = 2 | the data points are clustered into 2 groups. We would expect all the top values to belong to one cluster as they are close by. we can notice that few purple data points clustered with the yellow cluster as these data points are closer to centroid of yellow cluster compared to that of the purple cluster. We notice the 2 samples closer to each other form the single cluster when only 2 centroids are chosen as k-mean forms cluster centroids by averaging the data points. (best solution) |
| k = 3 | We would think 3 is the optimal number of cluster as we can notice 3 groups in the data. But when we apply 3 we can see 3 clusters but doesn't works as expected. The data set is nebulous and not spherical hence breaks the k-means assumption that the data points are spherical clusters. DBSCAN or Gaussian Mixture works well on such kind of data compared to k-means. |
| k = 6 | we can notice that each of the 3 groups that was previously formed gets split off into two making 6 clusters. We can't find an optimal solution to this data as it breaks the k-means assumption. |
| K = 9 | When 9 centroids where chosen the data from each of the 3 groups where separated into different centroids. For example, when k =3 centroid included data points from both the top strands. This stopped when k = 9. Now we could use the small clusters group them based on distance to get the correct grouping. |
| k= 30 | we have too many clusters when we increase the number of centroids and it's hard to make a conclusion on how these points are grouped making the data more granular. The data points belonging to each cluster are close by. |



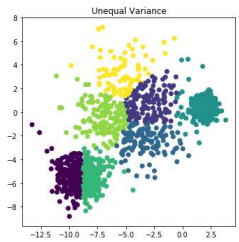

K =2



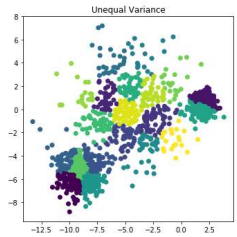

k =3




k= 4

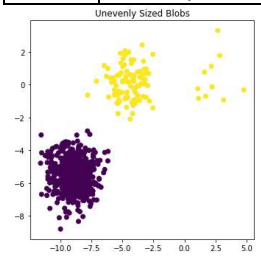



k =7

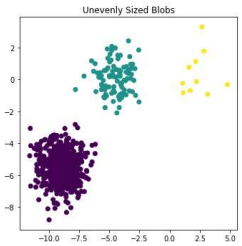



k=20

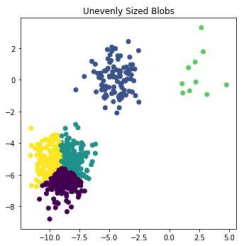
| | |
|---|---|
| k = 2 | the data points are clustered into 2 groups. As the k-means make certain assumptions on the data like the same variance in all directions or equal number of points in each cluster. We can notice the middle group of points being split among the 2 clusters. This is somewhat similar to what is expected. |
| k = 3 | We can see 3 clusters but doesn't works as expected. You would want every cluster (independently) to have the same variance in every variable, for the k-means to work correctly. (best solution) An algorithm like dbscan can perform better on such data when big epsilon is given. |
| k = 4 | k-means splits them incorrectly making 4 cluster based on the assumptions as stated above. |
| K = 7 | I choose this value because by combining few clusters at this stage we can easily distinguish the 3 clusters but on such kind of data we can use other algorithms to get a the results easily without having to combine. |
| k= 20 | we have too many clusters when we increase the number of centroids and it's hard to make a conclusion on how these points are grouped making the data more granular. |



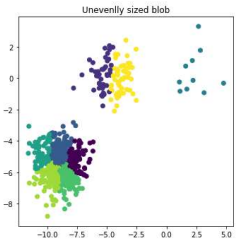

K =2



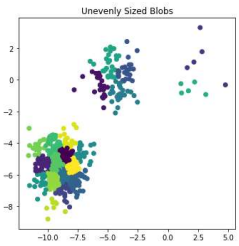

k=3




k=5




k=8



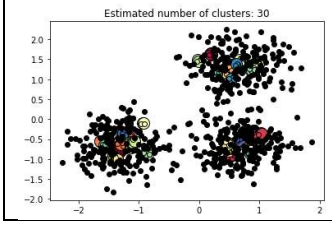

k=20

| | |
|---|---|
| k = 2 | the data points are clustered into 2 groups. The cluster result is close to what is expected. Some of the points which can be seen like outliers is grouped into the closest cluster. DBSCAN makes a better algorithm when it comes to detecting outliers. |
| k = 3 | We can see 3 clusters and works as expected. (best solution) |
| k = 5 | We can notice that the cluster with many samples is broken into smaller clusters when the number of centroid is increased. Because very often when you get two samples from the same cluster, it will get stuck in a minimum where this cluster remains split, and two other clusters merged instead. |
| K = 8 | As mention above on increasing the value only the huge highly dense samples break to form smaller cluster. When k reaches 8 we can notice the splitting in the secondly dense group. This is because every cluster (independently) should have the same variance in every variable. |
| k= 20 | we have too many clusters when we increase the number of centroids and it's hard to make a conclusion on how these points are grouped making the data more granular. |

b)

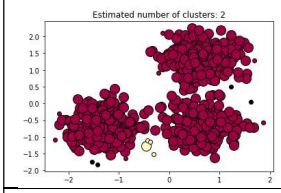
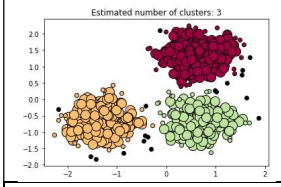
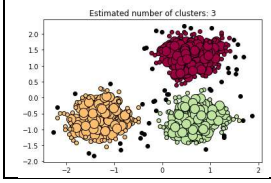
density-based approaches like DBSCAN model clusters as high-density clumps of points. The idea is that if a particular point belongs to a cluster, it should be near to lots of other points in that cluster which isn't taken into consideration under k-means.

The algorithm uses eps = 0.05 and notice what happens when min_samples is changed.

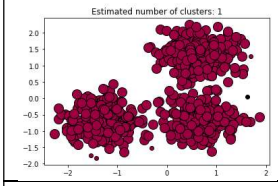
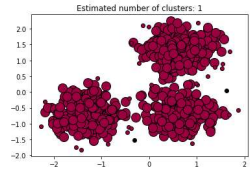
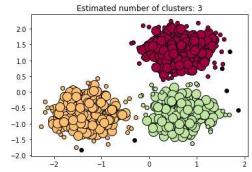
| min_samples | 0.05 |
|---|---|
| 4   | We notice that with a small number for min_sample we find too many clusters and as many more noise points. This is because we find many points that's surrounded by 4 points and thus forming a cluster. The epsilon value is very low for that dataset. If the data had very low variance then we can choose such small values. So we need to increase the distance to try getting a best results |

| | |
|---|---|
| 10 | No results as the algorithm cannot find as many data points in such small distance. |
| 20 | No results as the algorithm cannot find as many data points in such small distance. |

The algorithm uses eps = 0.3 and notice what happens when min_samples is changed.

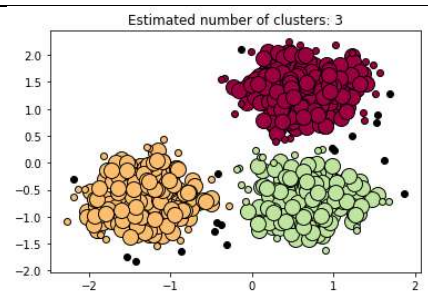
| min_samples | 0.3 |
|---|---|
| 4   | In this scenario we have pretty good eps, covers a larger region compared to that of the previous but on choosing a small number for min_samples we end up getting a single cluster as the data points of each cluster is not separated by a large distance from one another. |
| 10   | This is the default value given by DBSCAN program. With the sufficient eps value and min_samples we get a can see the 3 clusters. Few of the black dots here are the noise points. Homogeneity: 0.953, Completeness: 0.883 and V-measure: 0.917 |
| 20   | By increasing the min_samples we notice an increase in the number of noise points and decrease in the estimation matrices. A point here is considered into a cluster if it has 20 points nearby, twice the number used last time. Hence many of the points in the boundary are considered noise points. |

The algorithm uses eps = 0.4 and notice what happens when min_samples is changed.

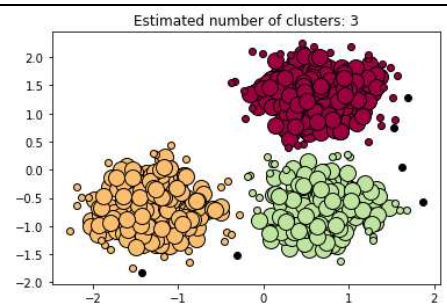
| min_samples | 0.4 |
|---|---|
| 4   | This is same as the previous case where we have a huge distance i.e. eps but it requires only one few points to form a cluster and hence leading to all points to form a single cluster. The other reason for this is the cluster groups are not separated by a huge distance. |
| 10   | Same as the above. Even though we have increased the min_samples the clusters are close enough to even find 10 points with this ep and hence not producing proper clustering. |
| 20   | If we further keep increasing the value of min_samples we start to find that the algorithm classifies the data points correctly. This is mainly because the algorithm checks for many points in the surrounding before forming a cluster. The boundary points don't not satisfy this condition hence leading to form proper clusters. |

With the default value we get 3 clusters and few noise points When I increased the eps and *min_samples* further the results improved. Few of the points that was classified as noise in with the default values where now in one of the cluster. The metrics values also increased.

Default value:

| | |
|---|---|
| *Default value: eps=0.3 min_samples= 10*<br><br>Estimated number of clusters: 3<br>Homogeneity: 0.953<br>Completeness: 0.883<br>V-measure: 0.917<br>Adjusted Rand Index: 0.952<br>Adjusted Mutual Information: 0.883<br>Silhouette Coefficient: 0.626 |  |

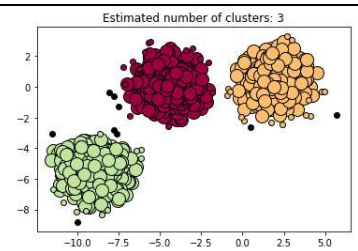
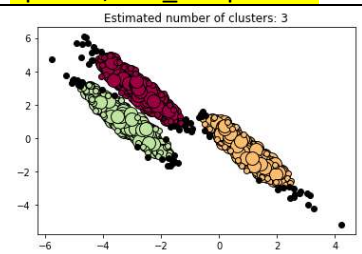
Interesting value:

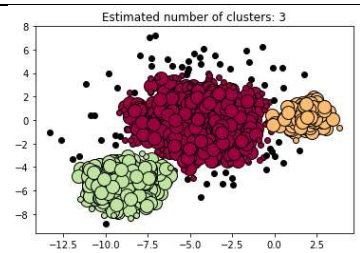
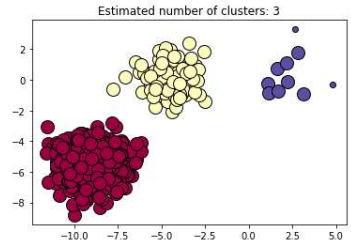
| | |
|---|---|
| *Default value: eps=0.4 min_samples= 20*<br><br>Estimated number of clusters: 3<br>Homogeneity: 0.945<br>Completeness: 0.913<br>V-measure: 0.929<br>Adjusted Rand Index: 0.960<br>Adjusted Mutual Information: 0.913<br>Silhouette Coefficient: 0.631 |  |

c)

In the below table the Best values and metrics used to evaluate the 4 data transformations.

**BEST VALUES:**

| | | |
|---|---|---|
| Number of blobs data: | Estimated number of clusters: 3<br>Homogeneity: 0.990<br>Completeness: 0.951<br>V-measure: 0.970<br>Adjusted Rand Index: 0.985<br>Adjusted Mutual Information: 0.951<br>Silhouette Coefficient: 0.722 |  <br>eps=0.9, min_samples=15 |
| Anisotropicly Distributed Blobs | Estimated number of clusters: 3<br>Homogeneity: 0.967<br>Completeness: 0.877<br>V-measure: 0.920<br>Adjusted Rand Index: 0.947<br>Adjusted Mutual Information: 0.877<br>Silhouette Coefficient: 0.455 |  <br>eps=0.3, min_samples=10 |

| | | |
|---|---|---|
| Unequal Variance | Estimated number of clusters: 3<br>Homogeneity: 0.940<br>Completeness: 0.847<br>V-measure: 0.891<br>Adjusted Rand Index: 0.912<br>Adjusted Mutual Information: 0.847<br>Silhouette Coefficient: 0.620 |  <br>eps=0.8, min_samples=10 |
| Unevenly Sized Blobs | Estimated number of clusters: 3<br>Homogeneity: 1.000<br>Completeness: 1.000<br>V-measure: 1.000<br>Adjusted Rand Index: 1.000<br>Adjusted Mutual Information: 1.000<br>Silhouette Coefficient: 0.745 |  <br>eps=2, min_samples=4 |

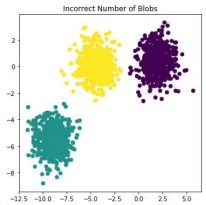
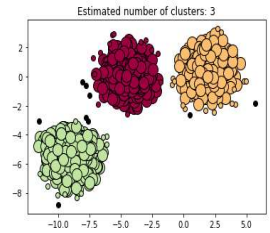
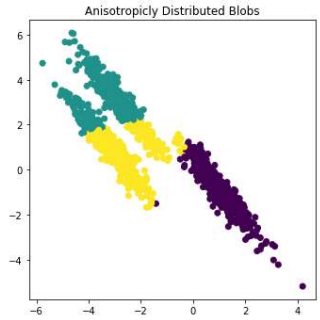
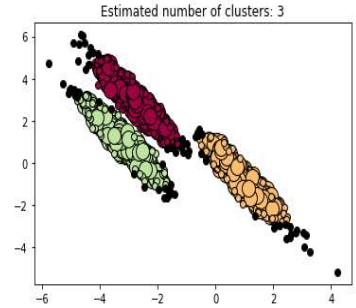
For case 1: Number of blobs data:
Experimenting with lower values for eps like 0.3 and min_num 10 gave around 20 clusters. This meant that the algorithm could find around 10 points in a distance of 0.3 but not many points where close by leading to multiple clusters, hence after multiple tries I found that a number as big as 0.9 gave best results. This means that the data points have a uniformly large varience. There are very few points which are very far from the clusters which is marked noise points, this option had a very good V-measure: 0.970.
Interesting point: When eps=0.2, min_samples=15 we found 3 clusters but in the visualization, we noticed maximum of noise points as the algorithm could hardly find points which was surrounded by 15 other data points in a distance of 0.2.

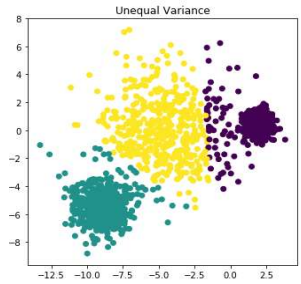
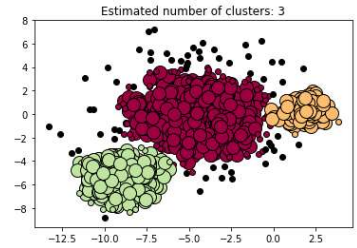
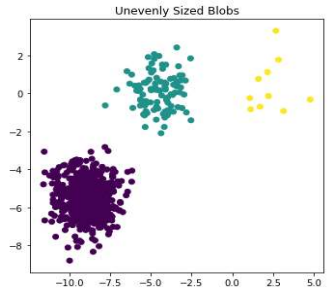
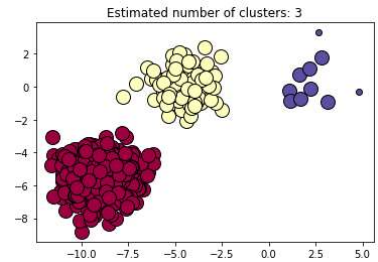
For case2: Anisotropicly Distributed Blobs:
For this example with a small eps 0.3 and min_sample of 10 we found the best result. Here the eps had to be small as the data points of two different clusters are close by. Changing the eps to a bigger number would lead to the all groups merging as the data points are close to each other. Here by keeping the eps constant if we decrease the min_sample to 5 we end getting more clusters as few of points which is supposed to noise is also formed a cluster as it requires very few points to form a cluster. If the epsilon is increased to 0.8 changing the min_sample from low value to a high value clustered all the data into 1 clusters. This is because of the nature of the dataset where the datapoints are close to each other.

For case3: Unequal Variance:
While applying dbscan on uneaqual variance data if we keep the eps to low and min_sample to any range from low to high we end up getting many clusters and also we can notice too many noise points. The clusters formed is mostly in the region where variance is low ie. the small 3$^{rd}$ cluster on the top. The middle blob of data where is variance is high usually ends up being noise points. By steadily increasing the eps we can start to see more more cluster points and noise points being reduced. On a value of eps = 0.8 with min_sample = 5 we find only 1 cluster. This is because when eps is high and we search for only 5 points around it to make it a cluster than in the given data transformation we end up getting a single cluster as these points are don't have a wide area of separation from one cluster to another.

For case4: Unevenly Sized Blobs:
When we use unevenly sized blobs with a small eps of 0.2 and min_samples of 4 we found that the algorithm found many clusters and blob with very small number of points is considered noise points. By increasing the eps and min_samples slowly we start to notice the algorithm is finding 2 clusters correctly but the smaller one is still notices as noise points. Just on further increasing of this value at a value of around eps = 2 and min_samples of 4 we get surprising results Estimated number of clusters: 3, Homogeneity: 1.000, Completeness: 1.000, V-measure: 1.000. Here in this case the clusters are far enough and the not that densely packed hence we need to keep the value of min_samples low and eps high.

Comparison of k-means and DBSCAN:

| | k-means | DBSCAN |
|---|---|---|
| Number of blobs data | On this data the algorithm works perfectly well when the number of cluster is chosen correctly. 3 for the above data which is the only parameter taken by the algorithm. As the data points are separate and follow few of the assumption made by the k-means algorithm like the clusters are usually spherical, all the clusters having equal variance. To choose the number of clusters we can use many methods like elbow or Silhouette.   | This density-based algorithm also clusters the data correctly when we choose the correct eps and min_sample for the sample. This algorithm works well for almost all kinds of data. Eps is the distance wherein the algorithm has to find at least min_sample of data points to add the corresponding point to the clusters.   |
| Anisotropicly Distributed Blobs | On this data the algorithm always fails to group into proper clusters. This is because of the nature of the data and also few assumptions made by the k-mean that the clusters are usually spherical and isotropic. In this algorithm once some initial centroids is chosen all the points are assigned to a cluster based on to which centroid the distance is minimized. Then the centroid points are recalculated which will be the average of the cluster points of that particular cluster. Hence the algorithm always fails on anisotropicly distributed blobs.   | This algorithm does a decent job on clustering the data when the parameters are chosen correctly. Here even though the data is Normally Distributed it's not isotropic hence a bad choice for k-means. The algorithm works well when the eps is kept low as the points belonging to different clusters are close enough to join into an other cluster. It also classifies few points which are very far from the clusters and that don't have many points around it as the noise points. The black dots we notice is are the noise points.   |
| Unequal Variance: | This kind of data cannot be handled well by the k-means because of the initial assumptions made. We can see 3 clusters but there are not well separating. Here the leftmost and rightmost clusters have a very low variance and the middle data have a high variance. | DNSCAN works perfectly good with this kind of data. Few points which are too far from the clusters are marked as noise points. We need set the eps for such data on a higher end in this case of 0.8 as the middle cluster data points have a high variance its won't be possible for the algorithm to find |

| | | |
|---|---|---|
| |  | min_sample with a small eps, leading to improper clustering.  |
| Unevenly Sized Blobs: | I think the k-mean works better for this kind of data as the unevenly sized are far apart from each other. The data also satisfies its normally distributed and isotopically. We just need to choose correct number of centroids for the algorithm to cluster correctly.  | DBSCAN algorithm detects the 3 clusters correctly but as these blobs are unevenly sized few of the data points of the smaller clusters gets recognized as noise points. Hence k-mean is a better choice for such kind of data.  |

d) The 20 Newsgroups data set is a collection of approximately 20,000 newsgroup documents, partitioned (nearly) evenly across 20 different newsgroups. The data is organized into 20 different newsgroups, each corresponding to a different topic. Some of the newsgroups are very closely related to each other (e.g. comp.sys.ibm.pc.hardware / comp.sys.mac.hardware), while others are highly unrelated (e.g misc.forsale / soc.religion.christian). The data folder contains subdirectory, each subdirectory in the bundle represents a newsgroup; each file in a subdirectory is the text of some newsgroup document that was posted to that newsgroup.

Homogeneity and Completeness – If you have pre-existing class labels that you're trying to duplicate with k-means clustering i.e. the ground truth, you can use two measures: homogeneity and completeness and V-Measure scores.

- Homogeneity means all of the observations with the same class label are in the same cluster in other words clustering result satisfies homogeneity if all of its clusters contain only data points which are members of a single class.
- Completeness means all members of the same class are in the same cluster in other words clustering result satisfies completeness if all the data points that are members of a given class are elements of the same cluster.
- The V-measure is the harmonic mean between homogeneity and completeness:

```
v = 2 * (homogeneity * completeness) / (homogeneity + completeness)
```

This metric is independent of the absolute values of the labels: a permutation of the class or cluster label values won't change the score value in any way.
The value of all the 3 parameters range from 0-1, 1 being the best possible solution

| | | |
|---|---|---|
| K= 4 | This value was chosen as the dataset here contains 4 categories of the dataset. So using 4 clusters should give a very high results on the performance measures. But when I looked at the results from the V-measure value this value of k turned out to be not working. | Homogeneity: 0.439<br>Completeness: 0.469<br>V-measure: 0.453 |
| K= 2 | This value was chosen to try out if we can increase the Homogeneity Completeness. We found that Completeness of the clusters increased but the Homogeneity fell down. | Homogeneity: 0.351<br>Completeness: 0.732<br>V-measure: 0.474 |
| K=20 | We could also increase the homogeneity of the cluster by increasing the number of clusters. As the clusters are now small enough get a high homogeneity but the V-measures still doesn't improve. | Homogeneity: 0.608<br>Completeness: 0.336<br>V-measure: 0.433 |

Increasing the number of number clusters gave us better Homogeneity whereas decreasing the number of cluster gave us better Completeness.

| | |
|---|---|
| Lsa (Latent Semantic Analysis) | The core idea is to take a matrix of what we have — documents and terms — and decompose it into a separate document-topic matrix and a topic-term matrix.<br>LSA assumes that words that are close in meaning will occur in similar pieces of text. A matrix containing word counts per paragraph is constructed from a large piece of text and SVD is used to reduce the number of rows while preserving the similarity structure among columns. Words are then compared by taking the cosine of the angle between the two vectors formed by any two rows. Values close to 1 represent very similar words while values close to 0 represent very dissimilar words. |
| Idf (Inverse document frequency) | IDF refers to the inverse of the number of documents our term appears in divided by the number of documents in our corpus. An *inverse document frequency* factor diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely. |
| hashing | is a fast and space-efficient way of vectorizing features, i.e. turning arbitrary features into indices in a vector or matrix. It works by applying a hash function to the features and using their hash values as indices directly, rather than looking the indices up in an associative array. |
| max num features | This is one parameter which we can choose in the tfidf vectorizer, if max_features is set to None, then the whole corpus is considered during the TF-IDFtransformation. Otherwise, if you pass, say, 5 to max_features, that would mean creating a feature matrix out of the most 5 frequent words accross text documents. |

For clusters = 4

| | |
|---|---|
| --use-hashing | Homogeneity: 0.435<br>Completeness: 0.491<br>V-measure: 0.462 |
| --no-idf | Homogeneity: 0.337<br>Completeness: 0.344<br>V-measure: 0.340 |
| n_features: 10 | Homogeneity: 0.044<br>Completeness: 0.045<br>V-measure: 0.044 |
| n_features: 100 | Homogeneity: 0.275<br>Completeness: 0.288<br>V-measure: 0.281 |
| n_features: 1000 | Homogeneity: 0.533<br>Completeness: 0.599<br>V-measure: 0.564 |

| | |
|---|---|
| n_features: 10000 | Homogeneity: 0.439<br>Completeness: 0.469<br>V-measure: 0.453 |
| Lsa:5 | Homogeneity: 0.586<br>Completeness: 0.629<br>V-measure: 0.607 |
| Lsa:10 | Homogeneity: 0.617<br>Completeness: 0.645<br>V-measure: 0.631 |
| Lsa:100 | Homogeneity: 0.481<br>Completeness: 0.490<br>V-measure: 0.486 |

Using hashing improved our results. Whereas removing the idf decreased all the 3 measures. Apart from having a very low value for n_features decreased all the measures keeping steadily increasing the value at n_features = 1000 we found our best result in the program. While further increasing n_features  we end up reducing the values. The same trend goes with the lsa, We get good measues in terms of all 3 values when lsa =10 but on further increasing the value of these metrics reduces.

Conclusion:

Here in this assignment we leant about when the k-means algorithm fails. We also learnt about few initial assumptions that the k-means algorithm makes while clustering the data points. Such as the data points should have equal variance. The data must be isotropic. We have to tune the parameters such as k which decided the number of cluster centroids to be used. To decide a best value for no. of centroids we can calculate using the elbow method.

The next algorithm we went through here was dbscan which was based on density of the points. This algorithm has 2 parameters that can be modified, the eps which specifies the distance around the point where in atleast min_samples must be found to form a cluster.This algorithm works well on all the data transformations on which the k-means failed when the right values for eps and min_samples where chosen. The algorithm apart from clustering also marks the noise points.

We then go through some of the evaluation metrics that can be used such as Homogeneity, Completeness, V-measure.

# PROGRAM

2c program:
For this program I have just added different transformations on the data and used it in the dbscan algorithm.
There are 2 parameters that needs to be modified when running on different data.
The eps and min_sample.
Here I have commented out the other transformations and the program shows the result for data which consists of unequally sized blob.

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt

from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

from sklearn.cluster import DBSCAN
from sklearn import metrics
from sklearn.preprocessing import StandardScaler

#plt.figure(figsize=(12, 12))

n_samples = 1500
random_state = 170
X, labels_true = make_blobs(n_samples=n_samples, random_state=random_state)


# Anisotropicly distributed data
transformation = [[0.60834549, -0.63667341], [-0.40887718, 0.85253229]]
#X_aniso = np.dot(X, transformation)
#X = np.dot(X, transformation) #change here

# Different variance
#X_varied, y_varied = make_blobs(n_samples=n_samples,
#                        cluster_std=[1.0, 2.5, 0.5],
#                        random_state=random_state)

#X, labels_true = make_blobs(n_samples=n_samples,
#                        cluster_std=[1.0, 2.5, 0.5],
#                        random_state=random_state)
#change above

# Unevenly sized blobs
#X_filtered = np.vstack((X[y == 0][:500], X[labels_true == 1][:100], X[labels_true == 2][:10]))
X = np.vstack((X[labels_true == 0][:500], X[labels_true == 1][:100], X[labels_true == 2][:10])) #change here
l5 = labels_true[labels_true == 0]
l5 = l5[:500]

l100 = labels_true[labels_true == 1]
l100 = l100[:100]
```

```
l10 = labels_true[labels_true == 2]
l10 = l10[:10]

labels_true = np.vstack((l5.reshape(500,1), l100.reshape(100,1), l10.reshape(10,1))) #change here
labels_true = labels_true[:610,0]

#X = StandardScaler().fit_transform(X)

# ##############################################################################
# Compute DBSCAN
db = DBSCAN(eps=0.2, min_samples=4).fit(X)
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_

# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)

print('Estimated number of clusters: %d' % n_clusters_)
print("Homogeneity: %0.3f" % metrics.homogeneity_score(labels_true, labels))
print("Completeness: %0.3f" % metrics.completeness_score(labels_true, labels))
print("V-measure: %0.3f" % metrics.v_measure_score(labels_true, labels))
print("Adjusted Rand Index: %0.3f"
    % metrics.adjusted_rand_score(labels_true, labels))
print("Adjusted Mutual Information: %0.3f"
    % metrics.adjusted_mutual_info_score(labels_true, labels))
print("Silhouette Coefficient: %0.3f"
    % metrics.silhouette_score(X, labels))

# ##############################################################################
# Plot result
#import matplotlib.pyplot as plt

# Black removed and is used for noise instead.
unique_labels = set(labels)
colors = [plt.cm.Spectral(each)
        for each in np.linspace(0, 1, len(unique_labels))]
for k, col in zip(unique_labels, colors):
   if k == -1:
      # Black used for noise.
      col = [0, 0, 0, 1]

   class_member_mask = (labels == k)

   xy = X[class_member_mask & core_samples_mask]
   plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
         markeredgecolor='k', markersize=14)

   xy = X[class_member_mask & ~core_samples_mask]
```

```
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=6)

plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```

2d. In this program only 4 clusters are taken as we have been not mentioned to change it. We set a random_state here to 0 so that we can get a consistent result. Apart from that I have changed the algorithm to use k-means instead of mini batch k-means. We need to pass in the arguments when using certain options like lsa idf which will be displayed once the program is run.

```
from __future__ import print_function

from sklearn.datasets import fetch_20newsgroups
from sklearn.decomposition import TruncatedSVD
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import HashingVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import Normalizer
from sklearn import metrics

from sklearn.cluster import KMeans, MiniBatchKMeans

import logging
from optparse import OptionParser
import sys
from time import time

import numpy as np


# Display progress logs on stdout
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(levelname)s %(message)s')

# parse commandline arguments
op = OptionParser()
op.add_option("--lsa",
              dest="n_components", type="int",
              help="Preprocess documents with latent semantic analysis.")
op.add_option("--no-minibatch",
              action="store_false", dest="minibatch", default=False,
              help="Use ordinary k-means algorithm (in batch mode).")
op.add_option("--no-idf",
              action="store_false", dest="use_idf", default=True,
              help="Disable Inverse Document Frequency feature weighting.")
op.add_option("--use-hashing",
              action="store_true", default=False,
```

```
        help="Use a hashing feature vectorizer")
op.add_option("--n-features", type=int, default=10000,
        help="Maximum number of features (dimensions)"
            " to extract from text.")
op.add_option("--verbose",
        action="store_true", dest="verbose", default=False,
        help="Print progress reports inside k-means algorithm.")

print(__doc__)
op.print_help()


def is_interactive():
    return not hasattr(sys.modules['__main__'], '__file__')

# work-around for Jupyter notebook and IPython console
argv = [] if is_interactive() else sys.argv[1:]
(opts, args) = op.parse_args(argv)
if len(args) > 0:
    op.error("this script takes no arguments.")
    sys.exit(1)


# ###############################################################################
# Load some categories from the training set
categories = [
    'alt.atheism',
    'talk.religion.misc',
    'comp.graphics',
    'sci.space',
]
# Uncomment the following to do the analysis on all the categories
# categories = None

print("Loading 20 newsgroups dataset for categories:")
print(categories)

dataset = fetch_20newsgroups(subset='all', categories=categories,
                             shuffle=True, random_state=42)

print("%d documents" % len(dataset.data))
print("%d categories" % len(dataset.target_names))
print()

labels = dataset.target
#true_k = np.unique(labels).shape[0]
true_k =15

print("Extracting features from the training dataset using a sparse vectorizer")
t0 = time()
```

```
if opts.use_hashing:
    if opts.use_idf:
        # Perform an IDF normalization on the output of HashingVectorizer
        hasher = HashingVectorizer(n_features=opts.n_features,
                                   stop_words='english', alternate_sign=False,
                                   norm=None, binary=False)
        vectorizer = make_pipeline(hasher, TfidfTransformer())
    else:
        vectorizer = HashingVectorizer(n_features=opts.n_features,
                                       stop_words='english',
                                       alternate_sign=False, norm='l2',
                                       binary=False)
else:
    vectorizer = TfidfVectorizer(max_df=0.5, max_features=opts.n_features,
                                 min_df=2, stop_words='english',
                                 use_idf=opts.use_idf)
X = vectorizer.fit_transform(dataset.data)

print("done in %fs" % (time() - t0))
print("n_samples: %d, n_features: %d" % X.shape)
print()

if opts.n_components:
    print("Performing dimensionality reduction using LSA")
    t0 = time()
    # Vectorizer results are normalized, which makes KMeans behave as
    # spherical k-means for better results. Since LSA/SVD results are
    # not normalized, we have to redo the normalization.
    svd = TruncatedSVD(opts.n_components)
    normalizer = Normalizer(copy=False)
    lsa = make_pipeline(svd, normalizer)

    X = lsa.fit_transform(X)

    print("done in %fs" % (time() - t0))

    explained_variance = svd.explained_variance_ratio_.sum()
    print("Explained variance of the SVD step: {}%".format(
        int(explained_variance * 100)))

    print()


# ##############################################################################
# Do the actual clustering

if opts.minibatch:
    km = MiniBatchKMeans(n_clusters=true_k, init='k-means++', n_init=1,
                         init_size=1000, batch_size=1000, verbose=opts.verbose)
else:
```

```
    km = KMeans(n_clusters=true_k, init='k-means++', max_iter=100, n_init=1,
                verbose=opts.verbose, random_state = 0)

print("Clustering sparse data with %s" % km)
t0 = time()
km.fit(X)
print("done in %0.3fs" % (time() - t0))
print()

print("Homogeneity: %0.3f" % metrics.homogeneity_score(labels, km.labels_))
print("Completeness: %0.3f" % metrics.completeness_score(labels, km.labels_))
print("V-measure: %0.3f" % metrics.v_measure_score(labels, km.labels_))
print("Adjusted Rand-Index: %.3f"
      % metrics.adjusted_rand_score(labels, km.labels_))
print("Silhouette Coefficient: %0.3f"
      % metrics.silhouette_score(X, km.labels_, sample_size=1000))

print()


if not opts.use_hashing:
    print("Top terms per cluster:")

    if opts.n_components:
        original_space_centroids = svd.inverse_transform(km.cluster_centers_)
        order_centroids = original_space_centroids.argsort()[:, ::-1]
    else:
        order_centroids = km.cluster_centers_.argsort()[:, ::-1]

    terms = vectorizer.get_feature_names()
    for i in range(true_k):
        print("Cluster %d:" % i, end='')
        for ind in order_centroids[i, :10]:
            print(' %s' % terms[ind], end='')
        print()
```