UCLA

ECE 219 - Large-Scale Data Mining: Models and Algorithms
Winter 2021

Project 2: Clustering

Authors:
Swapnil Sayan Saha (UID: 605353215)
Grant Young (UID: 505627579)

Question 1:

In this question, we are asked to extract numerical features from the textual data in the "20 Newsgroups dataset" to make the data suitable for use by clustering algorithms. The entire dataset contains roughly 18000 newsgroup posts assigned to one of 20 topics, with the training set containing 11314 textual posts evenly distributed across all topics. More specifically, the question asks to convert dataset (both training and test set combined) for 8 categories (4 from computer technology and 4 from recreational activity) into Term Frequency-Inverse Document Frequency (TF-IDF) matrix and report the shape of the resulting matrix. The shape is: **(7882, 23522)**

The steps undertaken for feature extraction are as follows:

- We removed headers and footers while importing the dataset using `remove` argument during dataset fetching.
- Use `CountVectorizer()` to convert the text (no lemmatization or stemming) into a *bag-of-words* matrix, which contains the frequencies of words in the vocabulary (use fit and transform on training set and transform on test set).
    - To omit common words in the English language (called stop-words) that do not provide any contextual information such as "and", "the", "I" etc., we set the argument `stopwords` to `"english"`.
    - To remove rare words, we set the argument `min_df` to 3. This removes all words that have a count of less than 3 in the training set. Decreasing the value of `min_df` increases the size of the resulting bag-of-words matrix.
- Convert the matrix of token counts to *TF-IDF matrix* (normalized matrix of counts based on importance of certain words) using `TfidfTransformer()` function (use fit and transform on training set and transform on test set). The TF-IDF matrix quantifies the importance of specific discriminating words (called keywords) in each document against words that are common across all documents within all classes due to the context of the classes. In other words, the TF-IDF matrix provides more useful and distinguishing features over bag-of-words for the classifier we want to use.

The tf-idf score for each term *t* in document *d*, with each element in bag-of-words matrix being tf(*t,d*) is:

$$\text{tf-idf}(t,d) = \text{tf}(t,d) \times \log\left(\frac{n}{\text{df}(t)}\right) + 1$$

where *n* = total number of documents, df(*t*) is the number of documents containing term *t*.

Question 2:

In this table, we are asked to report the contingency matrix of clustering results from K-Means clustering on the TF-IDF matrix with two classes (computer technology and recreational activity).

Clustering is referred to as the process of finding groups of data that shares similarity in the feature space, without using annotated classes.

The K-means algorithm looks for the centroid of K disjoint clusters of equal variance within the multimodal feature space of the dataset, allocating a certain data point to a cluster such that the in-cluster sum of squares (called inertia) is minimized and the insight that points within a cluster should be near the centroid. K-means (also called Lloyd's algorithm) is an iterative algorithm, which first involves initializing the centroids randomly followed by two repeated steps:

- Assign each sample to the nearest centroid.
- Create new centroids by taking the average value of all of the samples assigned to old centroids.

The algorithm stops when the centroids have stabilized or the defined number of iterations have been reached. Mathematically, the goal is to find $\boldsymbol{\mu}_k$ (center of cluster k) and r_{nk} such that the inertia is minimized:

$$\min J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} ||\mathrm{x}_n - \mu_k||^2$$

$$[r_{nk} = \begin{cases} 1, & \text{if } \mathrm{x}_n \text{ is assigned to cluster } k \\ 0, & \text{otherwise.} \end{cases}]$$

Here, x_n corresponds to the nth data point $n \in [1, N]$. K-means can be viewed as an expectation-maximization algorithm, where the covariance matrix is diagonal and all-equal. Advantages of k-means clustering include:

- The algorithm eventually converges given enough time.
- Simple and intuitive implementation with fast results delivery.
- Scales linearly in time for large datasets and adapts to new examples on-the-fly (unsupervised)

The disadvantages of k-means algorithm include:

- K-means assumes the clusters are spherical and evenly sized (isotropic), performing poorly for non-convex clusters, clusters of varying sizes and density.
- K-means assumes the clusters are univariant (Gaussian) in nature. As a result, centroids are affected by outliers and noise in the data.
- The algorithm does not scale well for high-dimensional data as the Euclidean distances converge to a constant value between all sample points.
- Initialization of K and centroid positions affect the final outcome, as the algorithm can converge to a local minima. One can use loss-vs-clusters plot to find the optimal value K, while initializing the centroids to be far from each other.

The contingency table is a generalization of the confusion matrix, with the rows showing the number of samples in ground truth clusters while the columns show the number of samples assigned to those clusters by the clustering algorithm. The diagonal (or sometimes anti-diagonal) entries represent the counts for correct conclusion (called `k-means++`).

Figure 1 shows the contingency table for K-Means clustering on the TF-IDF matrix with two classes (computer technology and recreational activity), with K set to 2 and centroids initialized with `k-means++`. Maximum number of iterations was set to 1000 with `n_init` set to 30, which indicates the number of times the algorithm reinitializes the centroid seed in `k-means++`. Higher values of `n_init` and number of iterations may improve clustering performance but require higher compute times. We can observe that the clustering algorithm did a good job correctly assigning most of the sample points to their respective clusters with the selected hyperparameters.

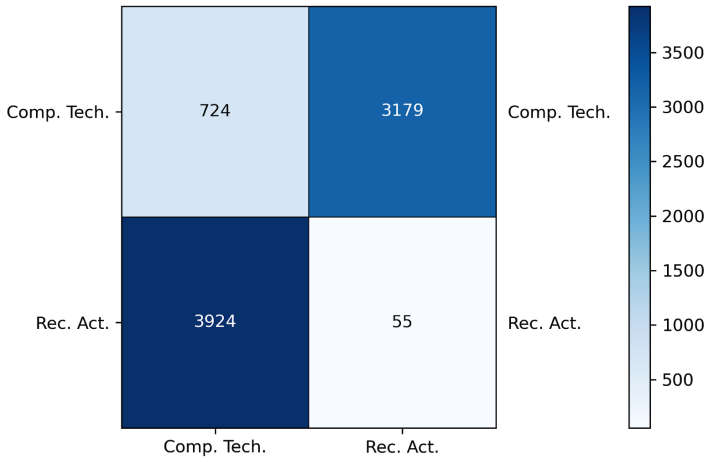


Figure 1: Contingency table for K-Means clustering on TF-IDF matrix with two classes.

Question 3:

In this question, we are asked to report 5 standard measures of clustering metrics used when labels are given for the results obtained in the last question. These metrics include:

- **Homogeneity**: A clustering result satisfies homogeneity if all of its clusters contain only data points which are members of a single class. Mathematically, homogeneity *h* is defined in terms of conditional entropy of labels *C* given cluster assignments *K* and is independent of absolute value of ground truth labels.

$$h = 1 - \frac{H(C|K)}{H(C)}$$

$$H(C|K) = -\sum_{c=1}^{|C|}\sum_{k=1}^{|K|}\frac{n_{c,k}}{n}\cdot\log\left(\frac{n_{c,k}}{n_k}\right)$$

$$H(X) = -\sum_{h=1}^{|H|}\frac{n_h}{n}\cdot\log\left(\frac{n_h}{n}\right)$$

$$n_{c,k} = n_k \cap n_c$$

  H(X) denotes the entropy of partition *X*, where *X* denotes non-overlapping groups of sample points. The information entropy is maximized when the sizes of the partitions are equal and minimized when some group within the partition takes up all the data points. The homogeneity score is maximized when each cluster K_i contains samples only from C_i, i.e. $H(C|K) = 0$

- **Completeness**: A clustering result satisfies completeness if all the data points that are members of a given class are elements of the same cluster. Mathematically, completeness is defined in terms of conditional entropy of clusters *K* given ground truth labels *C*.

$$c = 1 - \frac{H(K|C)}{H(K)}$$

  Completeness if maximized when each ground truth class C_i is part of some cluster K_i.

- **V-Measure:** V-measure is defined as:

$$\frac{(1+\beta)hc}{\beta(h+c)}$$

  $\boldsymbol{\beta}$ is usually considered 1. Compared to homogeneity and completeness, this metric is symmetric and is useful for measuring agreement between independent cluster assignment strategies on the same dataset.

- **Adjusted Rand Index (ARI):** ARI is a pairwise measure that calculates similarity between assigned clusters and ground truth labels. It takes into account all pairs of samples and counting pairs that are either assigned in the same cluster and the same class or different cluster and different class adjusted for chance. Mathematically, it is given by:

$$\mathrm{ARI} = \frac{\mathrm{RI} - \mathbf{E}(\mathrm{RI})}{\max(\mathrm{RI}) - \mathbf{E}(\mathrm{RI})}$$

$$\mathrm{RI} = \frac{TP + TN}{TP + FP + TN + FN}$$

- **Adjusted Mutual Information Score (AMI):** AMIS measures the mutual information (MI) between the cluster label distribution and the ground truth label distributions, adjusted for chance. It is defined as:

$$\mathrm{AMI} = \frac{\mathrm{MI(C,K)} - \mathbf{E}(\mathrm{MI(C,K)})}{\mathrm{avg(H(C),\ H(V))} - \mathbf{E}(\mathrm{MI(C,K)})}$$

  MI(C,K) measures the dependency of clustering labels on ground truth labels and vice versa.

The obtained metrics are:

- **Homogeneity: 0.581**
- **Completeness: 0.595**
- **V-measure: 0.588**
- **Adjusted Rand-Index: 0.644**
- **Adjusted Mutual Information Score: 0.588**

We can see that the homogeneity is ~60%, which indicates that 40% of the samples on average in a particular cluster are not from the assigned ground truth label. The completeness is also ~60%, which indicates that 60% of the samples for a given class label are part of the same cluster. The ARI and AMI scores indicate that the clustering distribution is dependent on or similar to the ground truth label distribution to moderate extent.

Question 4:

In this question, we are asked to reduce the dimensionality of the TF-IDF matrix and check what ratio of the variance of the original data is retained after the dimensionality reduction. As data moves into higher dimensions, the rapid increase in volume causes the data to become sparse in each dimension, causing the Euclidean distances to converge to a constant value between all sample points and degrading the clustering performance. This is referred to as the "Curse of Dimensionality". The question mentions using singular value decomposition (SVD) or Latent Semantic Indexing (LSI) for dimensionality reduction.

LSI or SVD reduces the rank of the feature matrix by multiplying the TF-IDF matrix $\mathbf{X}$ with the first k principal components in the feature space, represented in columns of matrix $\mathbf{V}_k$, which is found using singular value decomposition (SVD) of the TF-IDF matrix:

$$\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^T$$
$$\mathbf{X}_{\text{reduced}} = \mathbf{X}\mathbf{V}_k$$

The goal is to reduce the Frobenius norm of the difference between $\mathbf{X}$ and $\mathbf{U}_k \Sigma \mathbf{V}_k^{\mathrm{T}}$. k was chosen as 1000 as indicated in the question. To obtain the SVD of the TF-IDF matrix, we used the `TruncatedSVD()` function. We use fit and transform on the dataset. To calculate the variance retention percentage of top r (r is the same as k) principal components, we used the cumulative sum of the `explained_variance_ratio_`. Figure 2 shows the plot of plot of the percent of variance the top r principal components can retain versus r. We can see that as r increases, variance retention ratio increases uniformly as well.

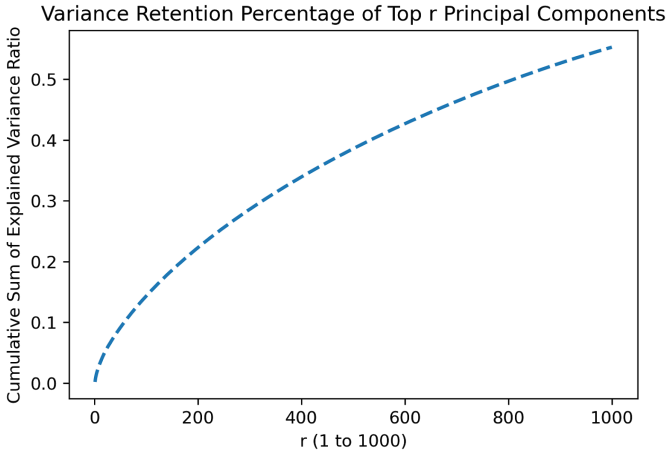


Figure 2: Percent of variance the top r principal components can retain versus r.

Question 5:

Similar to question 4, in this question, we are asked to reduce the dimensionality of the TF-IDF matrix, using both SVD (which is introduced in question 4) and Non-negative Matrix Factorization (NMF), with the goal of obtaining the ideal value of the number of principal components *r* for the dimension-reduced data in terms of the 5 metrics introduced in question 3. NMF attempts to find two non-negative matrices **W** and **H** whose product is as close to **X** as possible ($\{\mathbf{W} \in \mathcal{R}^{n\times r}, \mathbf{H} \in \mathcal{R}^{r\times m}\} | \mathbf{X} \in \mathcal{R}^{n\times m}$), with **W** being the low-rank feature matrix for **X**, where *r* = number of topics. **H** (coefficient matrix) provides weighing factors for all the topics in each document, while **W** (basis vectors) corresponds to the clusters discovered in the document. **W** and **H** are found by solving the following optimization problem:

$$\min_{\{\mathbf{W},\mathbf{H}\}\geq 0} ||\mathbf{X} - \mathbf{WH}||_F^2$$

We used the `NMF()` function to obtain **W**. The possible values of r included 1, 2, 3, 5, 10, 20, 50, 100 and 300. Figures 3, 4, 5, 6 and 7 show the plots of the 5 metrics vs *r* for both SVD and NMF for the aforementioned values of *r*, namely homogeneity, completeness, V-measure, ARI and AMI respectively.

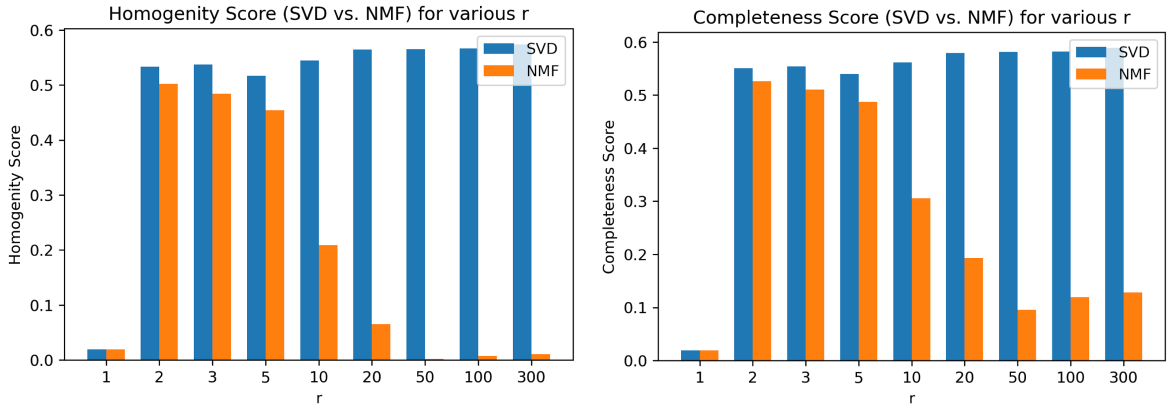


Figure 3 and 4: Plots of homogeneity and completeness for various values of *r* for both SVD and NMF.

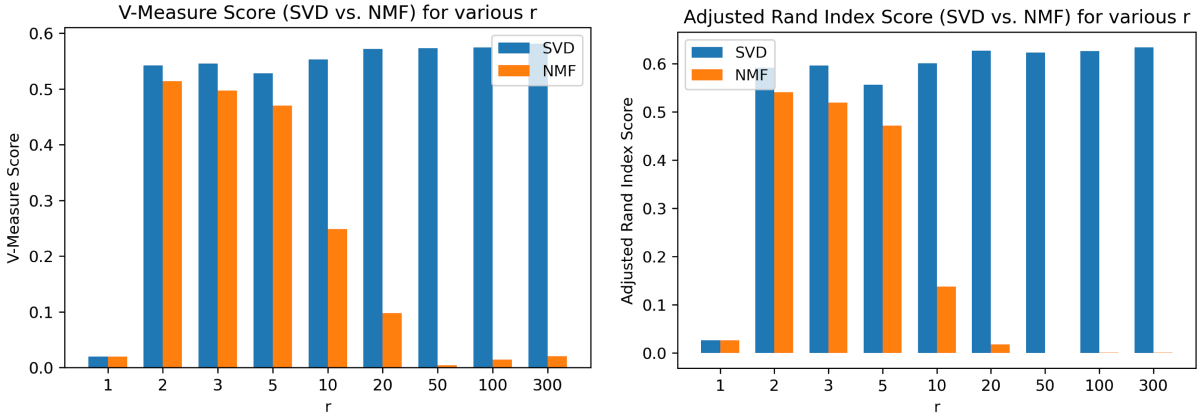


Figure 5 and 6: Plots of V-measure and ARI for various values of *r* for both SVD and NMF.

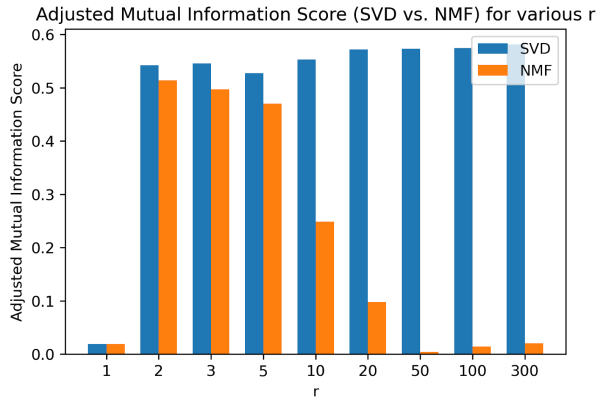


Figure 7: Plot of AMI for various values of *r* for both SVD and NMF.

From Figures 3 to 7, we can see that the relationship between clustering metric and number of principal components is non-monotonic. **We explain why this happens in the next question**. To choose the best value of *r*, we plotted the average value of all 5 metrics versus *r* on a single plot for both SVD and NMF, which is shown in Figure 8.

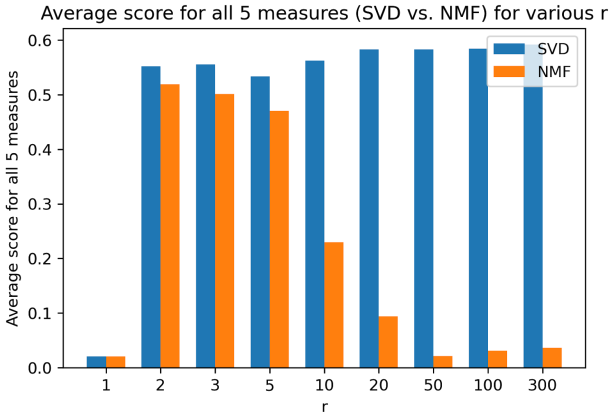


Figure 8: Plot of mean value of all 5 metrics for various values of *r* for both SVD and NMF.

From Figure 8, it is clear that the apparent best value of *r* for SVD is 300 and NMF is 2. However, we should take into account the fact that while *r* increases, although the amount of semantic and complex information and structure in the data available for clustering increases, the clustering performance drops because the Euclidean distances (which is the key metric for K-means clustering) converge to a constant value between all sample points (equidistant features). Thus, we need to choose a value of *r* that contains sufficient information but is also not too large as K-means does not perform well in high-dimensions. For SVD, we can see that the clustering performance does not increase significantly after $r = 20$. As a result, **a good choice for *r* for SVD is 20, while a good choice for *r* for NMF is 2**.

Question 6:

From Figures 3 to 8, we see that as *r* increases in general, the clustering metrics do not increase uniformly. If *r* is more, then the amount of semantic and complex information and structure in the data available for clustering increases. Theoretically, as more information is preserved, it should lead to better clustering in terms of separation and compactness. However, a larger value of *r* indicates a high-dimensional and noisy feature matrix, which degrades K-means clustering performance. This is because in higher dimensions, the rapid increase in volume causes the data to become sparse in each dimension, causing the Euclidean distances to converge to a constant value between all sample points. Since all feature points become equidistant and the inertia is not normalized, the clustering algorithm cannot find centroids with sufficient distance among them, leading to poor clustering. This is very evident for the metrics related to NMF, which initially increase but then drop rapidly.

In addition, we see that the performance of NMF is much more erratic than that of the SVD, which stabilizes after $r = 2$. This is because NMF only allows positive entries in the reduced-rank feature matrix while SVD has no such restriction. As a result, SVD is able to better represent the higher-dimensional feature matrix, providing a deeper factorization with lower information loss than NMF and causing the decomposition depth of NMF diminishes in high-dimensions (when *r* is large). Furthermore, SVD is much more deterministic than NMF, with a hierarchical and geometric basis ordered by relevance. NMF, on the other hand, does not consider the geometry in the feature space basis, which is essential for clustering in high dimensions. Since SVD produces a feature matrix with the most relevant features higher in the hierarchy, increasing the value of *r* does not cause significant drop or rise in clustering performance. From the K-means perspective, a feature matrix produced by SVD with higher values of *r* is not adding any significant distinguishable information, neither subtracting any important semantic information (due to noise) thanks to the ordering of the features. As a result, the clustering performance is nearly constant for values of *r* from 2 to 300, while the performance of NMF drops after $r = 2$. Last but not least, the results from SVD are unique while NMF is non-unique and stochastic, with no guarantees of convergence to the optimal feature matrix each time the function is called.

Question 7:

In this question, we are asked to visualize the clustered samples points compared to ground truth labels for SVD and NMF on 2D-plane, with our choice of *r*. Figure 9 shows the plot for SVD while Figure 10 shows the plot for NMF. To project onto the 2D plane, we plotted the first two principal components of the feature matrix generated by SVD and NMF.

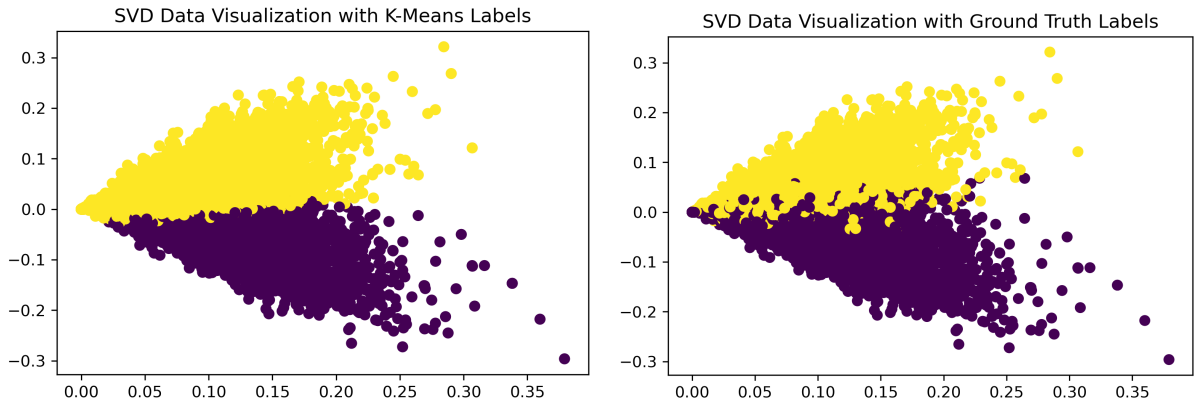


Figure 9: (Left) Plot of SVD generated feature matrix with labels generated by K-means clustering algorithm projected onto 2D plane (Right) Plot of SVD generated feature matrix with ground truth labels projected onto 2D plane (***r*** = 20).

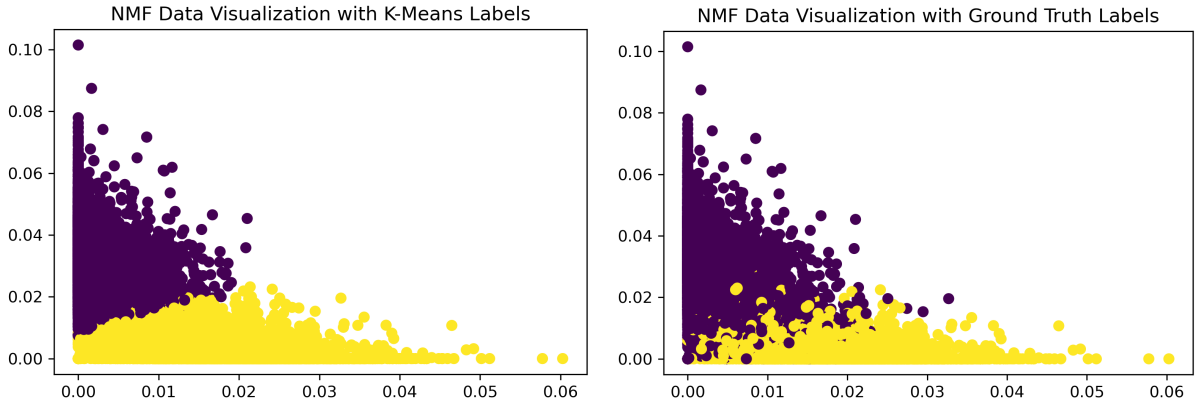


Figure 10: (Left) Plot of NMF generated feature matrix with labels generated by K-means clustering algorithm projected onto 2D plane (Right) Plot of NMF generated feature matrix with ground truth labels projected onto 2D plane (***r*** = 2).

Question 8:

There are several visual cues we can infer from the plots in Figure 9 and 10, which indicates the data distribution is not ideal for K-means clustering.

- K-means assumes the clusters are convex and isotropic. For both SVD and NMF, we do not see spherical distributions on the 2D plot, but rather irregularly shaped and elongated blobs. In addition, for NMF, we also observe unevenly sized blobs across the two classes.
- For optimal clustering, `k-means++` initializes the centroids to be far from each other. However, for both SVD and NMF, we can observe that there is significant overlap among the two clusters (closely packed) with minimal Euclidean distance between the centroids of the two clusters, leading to no definite decision boundary between the two clusters. This explains the low homogeneity and V-measure scores.
- K-means assumes the clusters are univariant (Gaussian) in nature. For both SVD and NMF, we see unequal variance for the two clusters, especially prominent in the feature distribution generated by the NMF with the yellow cluster much more compactly packed than the brown cluster. As a result, centroids are affected by outliers and noise in the data.

Question 9:

In this question, we are asked to perform clustering for all 20 categories in the "20 Newsgroups dataset". We took the following steps:

- Removed headers and footers.
- Used `CountVectorizer()` to convert the text into a bag-of-words matrix (no lemmatization or stemming used)
    - Argument `stopwords` set to "`english`".
    - Argument `min_df` set to 3
- Convert the matrix of token counts to TF-IDF matrix using `TfidfTransformer()`.
- Reduce dimensionality with SVD (question asked to either use SVD or NMF) for various values of *r* to find the best value of *r* for performing clustering on all 20 categories. The possible values of r included 1, 2, 3, 5, 10, 20, 50, 100 and 300.
- Use K-means clustering with 20 clusters. Maximum number of iterations was set to 1000 with `n_init` set to 30.

Table 1 shows the resulting clustering metrics for various values of *r*.

Table 1: Clustering metrics for various values of r for k-means clustering on all 20 categories via SVD

	r = 1	***r*** = 2	***r*** = 3	***r*** = 5	***r*** = 10	***r*** = 20	***r*** = 50 (best)	***r*** = 100	***r*** = 300
Homogeneity	0.024187148	0.212595089	0.247381314	0.321433377	0.324549957	0.333236617	0.345876618	0.320904819	0.320904819
Completeness	0.026467776	0.224768389	0.265937306	0.349255668	0.353869279	0.375569333	0.434467156	0.383267586	0.383267586
V-Measure	0.025276122	0.218512327	0.256323919	0.334767445	0.338576070	0.353138836	0.385143153	0.349324723	0.356896315
ARI	0.005242558	0.065567647	0.083921020	0.126562151	0.122662315	0.120532777	0.110426049	0.095640196	0.104059110
AMI	0.021952877	0.215913335	0.253821490	0.332523026	0.336342458	0.350916148	0.382906356	0.347024461	0.354568094

From table 1, we can see that the best average metric was obtained for *r* = 50. As a result, for this question, we plot the contingency matrix and report the 5 clustering metrics for **SVD with *r* = 50**. To mitigate mismatch between absolute labels and cluster labels, we used `scipy.optimize.linear_sum_assignment` to identify the best-matching cluster-class pairs, and permute the columns of the contingency matrix accordingly. Figure 11 shows the contingency matrix plot, while the 5 metrics are as follows:

Homogeneity (SVD, best r): 0.346

Completeness (SVD, best r): 0.434

V-measure (SVD, best r): 0.385

Adjusted Rand-Index (SVD, best r): 0.110

Adjusted Mutual Information Score (SVD, best r): 0.383

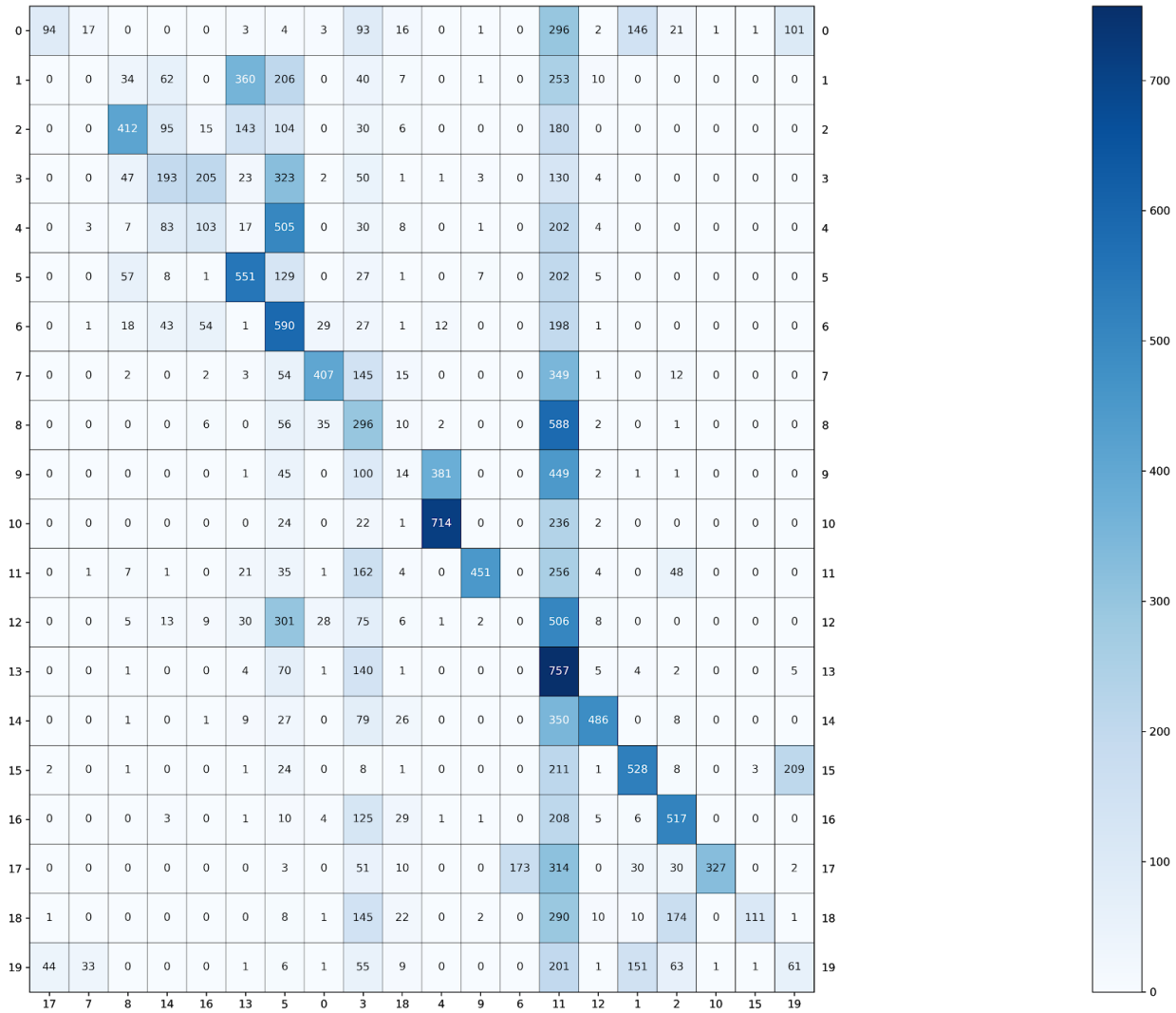


Figure 11: Contingency matrix visualization for K-means clustering (20 clusters) with SVD feature matrix ($r = 50$).

Question 10:

For this question, we are asked to check if Kullback-Leibler Divergence (KLD) cost function improves NMF performance. To do so, we first repeated the steps in Question 9, except extracting features using NMF for various values of *r*. We initially tried to find the best value of *r* with Frobenius norm, which we intended to use to find the metrics for KLD loss. Table 2 shows the resulting clustering metrics for various values of r for NMF with Frobenius norm.

Table 2: Clustering metrics for various values of r for k-means clustering on all 20 categories via NMF (Frobenius Norm)

	r = 1	***r*** = 2	***r*** = 3	***r*** = 5	***r*** = 10	*r* = 20 (best)	***r*** = 50	***r*** = 100
Homogeneity	0.02418293013	0.19105477902	0.21967746007	0.26683550314	0.28883124872	0.31148002169	0.25463244447	0.14029330462
Completeness	0.02646303476	0.20464240207	0.25707446624	0.28870335515	0.32061307797	0.37240862605	0.34717245207	0.19268911998
V-Measure	0.02527165677	0.19761530167	0.23690922962	0.27733903355	0.30389347020	0.33923021621	0.29378747376	0.16236889042
ARI	0.00524174810	0.05763353989	0.06731888280	0.08697997264	0.10180156132	0.09567574557	0.06671757556	0.02655053209
AMI	0.02194840487	0.19492006697	0.23421564285	0.27490744181	0.30152097794	0.33688862108	0.29109625864	0.15918079718

From Table 2, we can see that the best average metric was obtained for r = 20. As a result, for this question, we chose ***r* = 20**. Afterwards, we calculated the 5 clustering metrics but this time we updated the beta loss to KLD and set the solver to multiplicative update, while choosing *r* = 20. The 5 clustering metrics are as follows:

Homogeneity (NMF KL, best r): 0.424
Completeness (NMF KL, best r): 0.456
V-measure (NMF KL, best r): 0.439
Adjusted Rand-Index (NMF KL, best r): 0.214
Adjusted Mutual Information Score (NMF KL, best r): 0.437

Comparing the shaded portion of Table 2 with the 5 metrics we obtained above, we see that **KLD cost function improves the clustering performanc**e. This can be attributed to various reasons:

- Text documents are generally sparse in the feature space. NMF with KL divergence are able to generate features that account for the sparseness, describing the stochastic variations and outliers better than NMF with Frobenius norm. Such feature matrices provide more compact representation of the appearance of attributes over latent components in the feature space as well as provide concise interpretability of the contribution of latent components over samples.

- NMF with Frobenius norm suffers from scale invariance issues due to its dependence on distance based learning. Since KLD is probabilistic in nature, the parameters for NMF with KLD are probabilistic mixture models, invariant across scales of data or outliers.
- NMF with KLD is said to be a probabilistic version of LSI/LSA (PLSA). We mentioned earlier that LSA (or SVD) has several advantages over NMF with Frobenius Norm:
    - LSA has hierarchical and geometric basis ordered by relevance, producing a feature matrix with the most relevant features higher in the hierarchy, while NMF has no hierarchical extensions. However, since NMF with KLD is essentially PLSA, it features hierarchical extensions for organizing text rather than being limited by flat vectors for NMF with Frobenius norm.
    - NMF with Frobenius Norm is much more erratic in nature than LSA. However, NMF with KLD is much less sensitive parameter initialization with a stabilized parameter estimation paradigm thanks to use of tempered expectation maximization.
- KLD (derived from Shannon entropy) can be thought of as a measure of information loss (entropy loss) between feature distribution and ground truth distribution, which guarantees stronger convergence over distance based losses such as Frobenius norm or mean-squared error. In fact, entropy derived losses are very common for use in deep-learning due to their ability to deal with non-linear data distributions.

Question 11:

For this question, we are asked to use Uniform Manifold Approximation and Projection (UMAP) for dimension reduction and find the ideal number of principal components for two distance metrics, namely euclidean and cosine distance.

UMAP constructs graphical representation of the data in high-dimensions (Cech complex) and attempts to optimize a low-dimensional graphical embedding to be as geometrically similar as possible to the high-dimensional graph. The high-dimensional graph is constructed using a weighted graph with the edges representing probability of association of two samples. The association radius is locally determined via the distance between k nearest neighbors to a sample point, with decreasing connection likelihood as the radius increases (fuzziness). The number of neighbours controls how UMAP balances local versus global structure and whether it weighs the big picture vs. local connections during graph construction. UMAP then projects the data into lower dimensions using a force-directed graph layout algorithm, with a hyperparameter controlling how tightly or loosely lumped the embedding points are going to be.

Table 3 and 4 shows the resulting 5 clustering metrics for various values of r (r denotes the number of principal components) for UMAP with the two distance metrics, performing K-means clustering on all 20 categories.

Table 3: Clustering metrics for various values of r for k-means clustering on all 20 categories via **UMAP (Euclidean)**

	$r = 1$	$r = 2$	$r = 3$	$r = 5$	$r = 10$	$r = 20$	$r = 50$ (best)	$r = 100$
Homogeneity	0.01262029309	0.00841997753	0.00979411694	0.01367909812	0.01346768229	0.01382494193	0.01458029362	0.01168241355
Completeness	0.01417411676	0.00927332857	0.01024766066	0.01414939482	0.01448481977	0.01456512704	0.01572884899	0.01232486456
V-Measure	0.01335215135	0.00882607442	0.01001575697	0.01391027250	0.01395774520	0.01418538544	0.01513280924	0.01199504284
ARI	0.00074589480	0.00104448497	0.00142893515	0.00217714110	0.00265081579	0.00294649275	0.00296173125	0.00215407157
AMI	0.00994204202	0.00545937114	0.00675219998	0.01064441623	0.01069524785	0.01093182608	0.01183212738	0.00873190262

Table 4: Clustering metrics for various values of r for k-means clustering on all 20 categories via **UMAP (Cosine)**

	$r = 1$	$r = 2$	$r = 3$	$r = 5$ (best)	$r = 10$	$r = 20$	$r = 50$ (best)	$r = 100$
Homogeneity	0.38764015955	0.54149373695	0.55034537268	0.56072894937	0.52872662538	0.53961557107	0.54752802751	0.54847306864

Completeness	0.40129324010	0.55339804649	0.56564158284	0.56601202979	0.56553651942	0.57115748526	0.58327815865	0.59104813767
V-Measure	0.39434856146	0.54738117638	0.55788864945	0.56335810389	0.54651244879	0.55493869035	0.56483797772	0.56896525310
ARI	0.25045111085	0.41412727593	0.43361466771	0.44732645053	0.40713388685	0.41377800373	0.42472435424	0.41705979104
AMI	0.39234263463	0.54589185757	0.55642929878	0.56194183364	0.54498372418	0.55344624448	0.56337369316	0.56750626943

From Table 3, we see that a good choice of *r* for Euclidean UMAP is 50, while from Table 4, we see that a good choice of *r* for Cosine UMAP is 5. The reported metrics are as follows:

Homogeneity (UMAP (euclidean), best r): 0.014580293627545 61
Completeness (UMAP (euclidean), best r): 0.015728848995795085
V-measure (UMAP (euclidean), best r): 0.01513280924056315
Adjusted Rand-Index (UMAP (euclidean), best r): 0.0029617312569419113
Adjusted Mutual Information Score (UMAP (euclidean), best r): 0.011832127381706347

Homogeneity (UMAP (cosine), best r): 0.5607289493767216
Completeness (UMAP (cosine), best r): 0.5660120297949187
V-measure (UMAP (cosine), best r): 0.5633581038914941
Adjusted Rand-Index (UMAP (cosine), best r): 0.44732645053169007
Adjusted Mutual Information Score (UMAP (cosine), best r): 0.5619418336479003

From the metrics, we can see that UMAP with cosine distance metric performs much better clustering than UMAP with Euclidean distance. This is expected because cosine similarity is not affected by the magnitude of the vectors, meaning that the length of the documents does not affect the distance metric, but rather it associates clusters based on angle between sample points. This helps correct the fact that higher frequency of words in a document does not necessarily mean it belongs to a certain class given those words, it could also mean the document is very long. In addition, in high dimensions, the rapid increase in volume causes the data to become sparse in each dimension, causing the Euclidean distances to converge to a constant value between all sample points. Since all feature points become equidistant, UMAP cannot find distinguishable clusters within the data. Thus, for textual clustering, cosine similarity is the ideal metric over L2 distances.

The contingency matrix for Euclidean UMAP ($r = 50$) and Cosine UMAP ($r = 5$) are shown in Figure 12 and Figure 13 respectively. We provide an explanation of the contingency matrices in the next question.

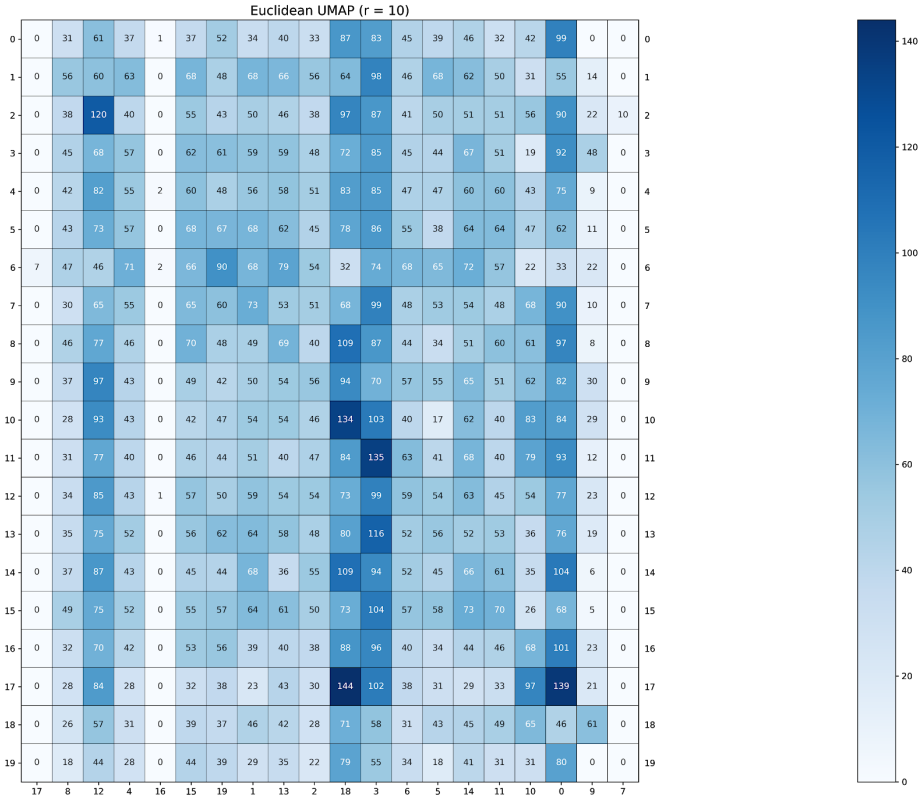


Figure 12: Contingency matrix visualization for K-means clustering (20 clusters) with Euclidean UMAP feature matrix ($r = 50$).

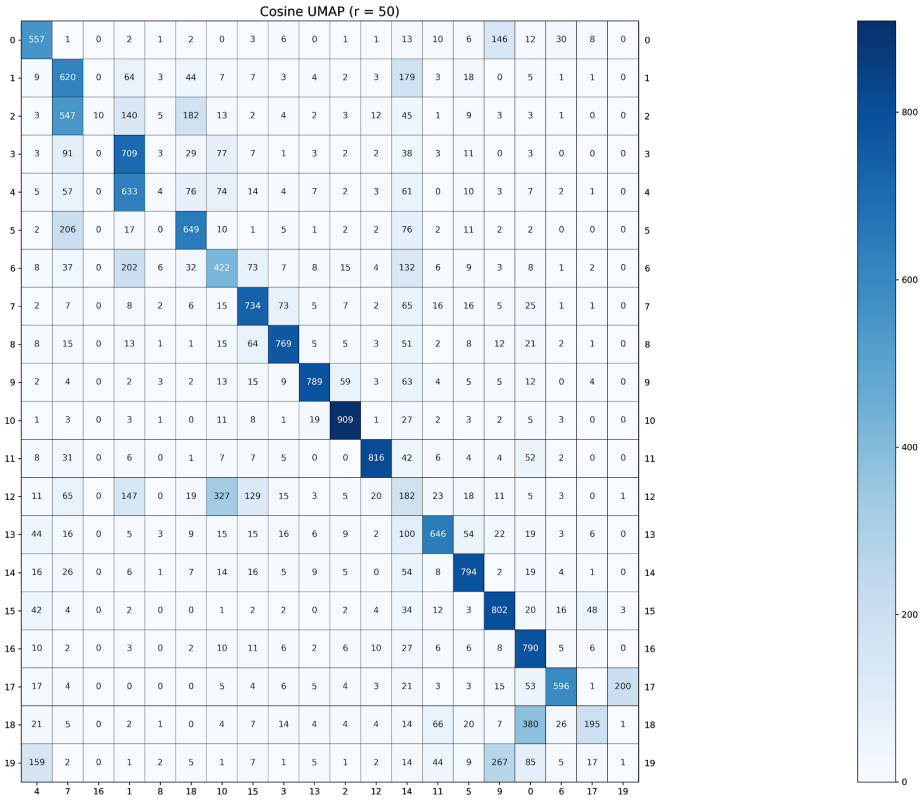


Figure 13: Contingency matrix visualization for K-means clustering (20 clusters) with Cosine UMAP feature matrix ($r = 5$).

Question 12:

Since Euclidean UMAP's contingency table is highly non-diagonal and stochastic (Figure 12), we instead explain the contingency matrix for Cosine UMAP (Figure 13). We see that some of the ground truth clusters are being lumped to one cluster. The problematic clusters include rec.motorcycles, sci.space, talk.politics.guns, talk.politics.mideast and talk.religion.misc. This is probably due to use of common words within these classes, exacerbated by the fact that headers and footers which may aid in providing semantic information about the topic of the newsgroup posts are missing. In addition, from a machine learning perspective, K-means also has several shortcomings:

- Inability of K-means to handle clusters of different sizes and densities.
- Inability of K-means to handle non-globular, non-isotropic and non-convex clusters.
- Does not work for clusters that have vague notion of centroids.
- Sensitivity to noise and outliers.

Question 13:

In this question, we are asked to use agglomerative clustering on UMAP transformed data for all 20 categories using 20 clusters and compare the performance of "ward" and "single" linkage criteria. For this question, we used UMAP with cosine distance metric and $r = 5$ to reduce the dimensionality of the TF-IDF matrix.

Agglomerative clustering is a form of hierarchical clustering following a bottom-up approach. Each data point is initially considered a leaf or individual cluster, and the distance between such clusters are calculated. The two clusters with the shortest distance are merged (called a node). The newly formed clusters perform the distance calculation once again with other clusters. The process is repeated until all the data points are assigned to one cluster called the root, resulting in a tree-like representation of the elements called a dendrogram. The linkage criteria determines the metric used for the merge strategy. The five clustering metrics for "ward" and "single" linkage criteria are as follows:

Agglomerative Clustering, Ward - Homogeneity: 0.539
Agglomerative Clustering, Ward - Completeness: 0.560
Agglomerative Clustering, Ward - V-measure: 0.550
Agglomerative Clustering, Ward - Adjusted Rand-Index: 0.406
Agglomerative Clustering, Ward - Adjusted Mutual Information Score: 0.548

Agglomerative Clustering, Single - Homogeneity: 0.016
Agglomerative Clustering, Single - Completeness: 0.410
Agglomerative Clustering, Single - V-measure: 0.032
Agglomerative Clustering, Single - Adjusted Rand-Index: 0.000
Agglomerative Clustering, Single - Adjusted Mutual Information Score: 0.027

From the above metrics, we see that single linkage criteria performs significantly worse than ward linkage criteria. The single linkage criterion (nearest neighbour) defines the distance as the minimum distance between clusters data points, while the ward linkage criterion (minimal increase of sum-of-squares method) minimizes the sum of squared differences within all clusters. As a result, although single linkage is fast, efficient and works well on non-elliptical data, single linkage clustering is not robust to noisy data or outliers and leads to formation of long snakelike chains (as agglomerative clustering follows "rich get richer" strategy), which causes data points to be erroneously clustered in one group. Ward linkage criterion (equivalent to K-means clustering), on the other hand, encourages formation of spherically tightly bound clusters and works well in presence of noise.

Question 14:

In this question, we are asked to perform hyperparameter tuning and report clustering metrics for DBSCAN (Density-Based Spatial Clustering of Applications with Noise) and HDBSCAN (Hierarchical DBSCAN) clustering algorithms on UMAP transformed data, with `min_cluster_size` hyperparameter for HDBSCAN set to 100. For this question (similar to last question), we used UMAP with cosine distance metric and r = 5 to reduce the dimensionality of the TF-IDF matrix before feeding the matrix to DBSCAN and HDBSCAN algorithm.

The DBSCAN algorithm is based on the heuristic that similar samples in the feature space are densely packed (core samples), separated by areas of low density. As a result, DBSCAN is not sensitive to the shape of clusters and works well with non-isotropic and non-convex clusters and works well with large noisy datasets with stable performance across runs. DBSCAN groups together points that are close to each other based on a distance metric (usually Euclidean), the minimum number of samples required to form a cluster or to be considered a dense region and a cluster selection epsilon, which specifies the maximum distance allowed between two points for them to be considered part of the same cluster. DBSCAN marks points in low-density regions as outliers or noise and does not group them in any cluster (often leading to -1 cluster label in the contingency table which indicates "noise"). DBSCAN is related to agglomerative clustering in its manifold following behavior, in the sense it uses single linkage criteria to form dendrograms after transforming the feature space according to data density. However, agglomerative clustering is a form of partitioning and does not take into account data density.

HDBSCAN extends DBSCAN by converting it into a hierarchical clustering algorithm, with the goal of allowing varying density clusters. The algorithm is similar to DBSCAN, except that the dendrograms are cut at varying heights to select stable clusters, resulting in a smaller tree with fewer clusters that lose points.

The two most important hyperparameters for both algorithms are cluster selection epsilon and minimum samples per cluster as discussed. If epsilon is too small, most of the samples will be treated as outliers, while a large value erroneously merges different clusters together. Large values of minimum samples per cluster are helpful for rejecting noise in the data. The chosen values for hyperparameter tuning include:

- Cluster selection epsilon, e = [0.01,0.05, 0.1, 0.15, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0,3.0,5.0,10.0,30.0,50.0]
- Minimum samples per cluster, m = [5,15,30,60,100,200,500,1000,3000]

We used nested for loops to perform hyperparameter tuning. **The best value of e and m were found to be 0.5 and 60 respectively for DBSCAN, whereas the best value of e and m were found to be 0.4 and 30 for HDBSCAN.** The obtained metrics for the best hyperparameters for DBSCAN and HDBSCAN are as follows:

- **Homogeneity (DBSCAN, best hyperparameters): 0.46122031931292645**
- **Completeness (DBSCAN, best hyperparameters): 0.5851789552890995**
- **V-measure (DBSCAN, best hyperparameters): 0.515857438292457**
- **Adjusted Rand-Index (DBSCAN, best hyperparameters): 0.311695108903137**
- **Adjusted Mutual Information Score (DBSCAN, best hyperparameters): 0.5144464043441347**

- **Homogeneity (HDBSCAN, best hyperparameters): 0.4419241631830412**
- **Completeness (HDBSCAN, best hyperparameters): 0.5742278618978046**
- **V-measure (HDBSCAN, best hyperparameters): 0.49946299585514187**
- **Adjusted Rand-Index (HDBSCAN, best hyperparameters): 0.2839171766231235**
- **Adjusted Mutual Information Score (HDBSCAN, best hyperparameters): 0.4984069795137075**

From the metrics, we see that both HDBSCAN and DBSCAN performed very similarly without significant differences in the 5 clustering metrics. This is expected because the only difference between HDBSCAN and DBSCAN is the ability to deal with varying density clusters, which if absent in the density space, does not yield any additional performance when using HDBSCAN over DBSCAN.

Question 15:

In this question, we are asked to interpret the contingency matrices for the DBSCAN and HDBSCAN. Figure 14 and 15 shows the contingency matrices for best models obtained in the last question.

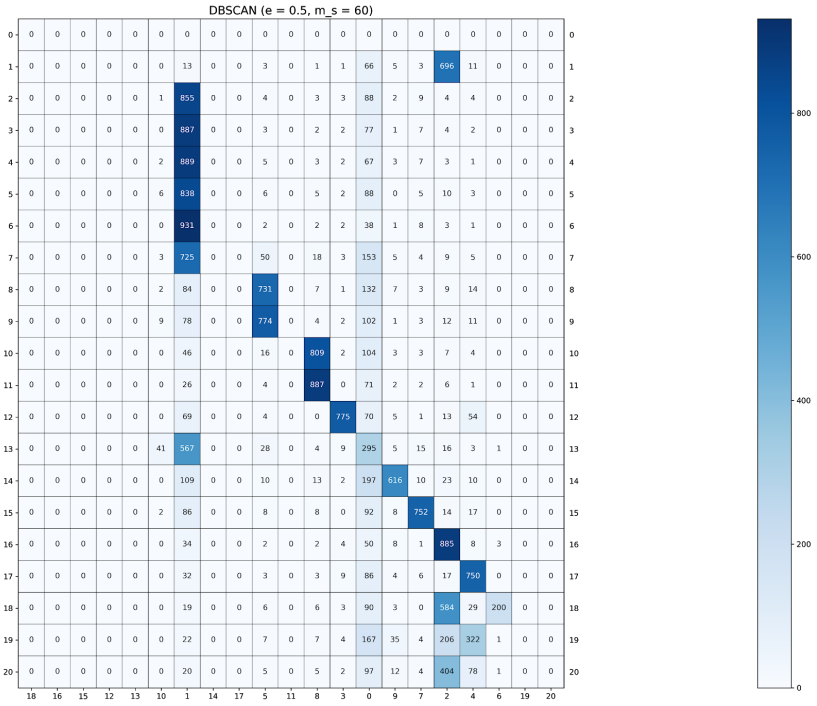


Figure 14: Contingency matrix visualization for DBSCAN clustering, e = 0.5 and m = 60.

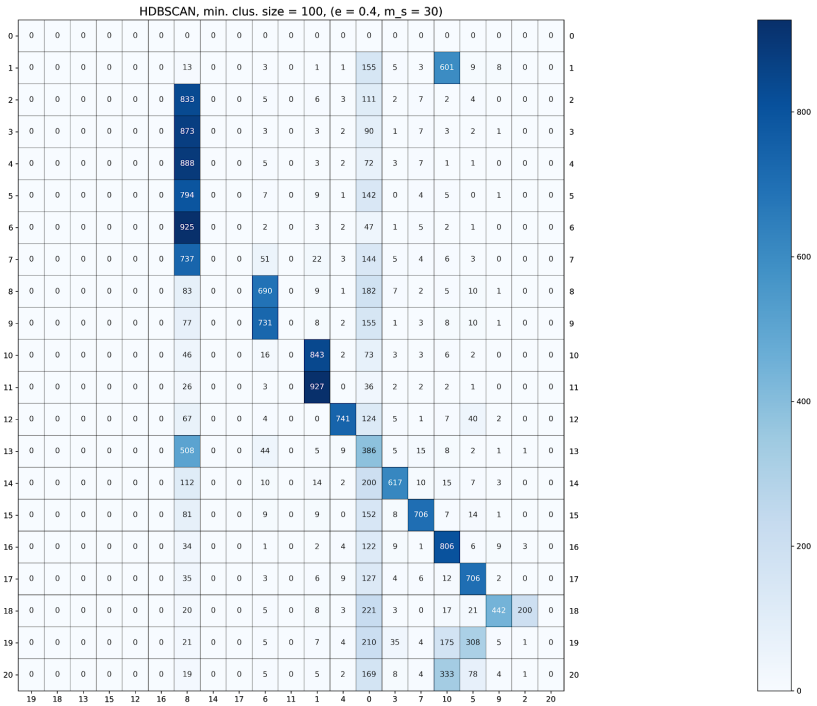


Figure 15: Contingency matrix visualization for HDBSCAN clustering, e = 0.4 and m = 30.

From the contingency matrices, we can see that DBSCAN produced a total of 10 major clusters (total 11), while HDBSCAN has produced a total of 11 major clusters. A lot of the missing ground truth clusters have been lumped into other clusters as the clusters are actually a tree and not flat. The "-1" in clustering labels refers to outliers or noisy samples that have not been classified into any cluster by the algorithms. The lumping of clusters is probably due to excessive smoothing caused by sensitivity to the hyperparameters without altering the minimum cluster size, which has caused loss of clusters of low density. In addition, both algorithms fail to identify clusters if they are varying wildly or are sparse in high dimensions, which is indeed the case for textual data. A good option might be to experiment with the minimum cluster size hyperparameter to see if altering it leads to formation of more microcluster for low-density classes, especially for HDBSCAN.

Question 16:

In this question, we are asked to present a clustering pipeline for BBCNews Dataset. The dataset contains 2225 articles, each labeled under one of 5 categories: business, entertainment, politics, sport or tech. The dataset is broken into 1490 records for training and 735 for testing.

Data Acquiring

To acquire data, we simply went to the link provided in the question and downloaded the CSV file containing the training data with labels. We used pandas to load the dataset into memory, importing only the "Text" and "Category" columns. The "Category" column, which corresponds to class labels, were then stored in a separate pandas dataframe, which was then factorized to convert to numerical class labels. To convert the dataset from an array of arrays to array of strings (list), we used the following line:

```
X_BBC=["".join(text) for text in BBC_dataset_train['Text'].values]
```

The resulting dimensions are as follows:

Size of labels (factorized to integer): (1490,)
Size of training set (list): 1490

Feature Engineering

We followed similar feature extraction pipeline as the past questions:

- Used `CountVectorizer()` to convert the text into a bag-of-words matrix (no lemmatization or stemming used)
    - Argument `stopwords` set to "`english`".
    - Argument `min_df` set to 3
- Convert the matrix of token counts to TF-IDF matrix using `TfidfTransformer()`.

For dimensionality reduction, we wanted to test 4 techniques:

- SVD
- NMF (frobenius norm)
- UMAP (Euclidean)
- UMAP (Cosine Similarity)

The number of principal components ***r*** was treated as a hyperparameter, consisting of [1,2,3,5,10,20,50,100] and the best value of ***r*** was found using K-Means.

Clustering

For clustering, we chose to test the following algorithms:

- K-Means (on all four dimension reduction techniques) with 5 clusters, maximum number of iterations was set to 1000 with n_init set to 30. The number of principal components r was treated as a hyperparameter, consisting of [1,2,3,5,10,20,50,100] and the best value of r was found using K-Means.
- Agglomerative clustering (on UMAP (cosine) dimension reduction technique with the best value of r found using K-Means), hyperparameter: linkage criteria (ward versus single).
- DBSCAN (on UMAP (cosine) dimension reduction technique with the best value of r found using K-Means), cluster selection epsilon and minimum samples per cluster were treated as hyperparameters and optimal values found via nested for loop search:
    - Cluster selection epsilon, e = [0.01,0.05, 0.1, 0.15, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0,3.0,5.0,10.0,30.0,50.0]
    - Minimum samples per cluster, m = [5,15,30,60,100,200,500,1000,3000]
- HDBSCAN (on UMAP (cosine) dimension reduction technique with the best value of r found using K-Means), minimum cluster size was set to 100, cluster selection epsilon and minimum samples per cluster were treated as hyperparameters and optimal values found via nested for loop search:
    - Cluster selection epsilon, e = [0.01,0.05, 0.1, 0.15, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0,3.0,5.0,10.0,30.0,50.0]
    - Minimum samples per cluster, m = [5,15,30,60,100,200,500,1000,3000]

Performance Evaluation:

To evaluate performance, we used the 5 clustering metrics, namely homogeneity, completeness, V-measure, ARI and AMI, as well as contingency matrix plots.

Figures 16, 17, 18, 19 and 20 show the plots of the 5 metrics vs r for SVD, NMF, UMAP (euclidean) and UMAP (cosine) for the aforementioned values of r for K-means clustering. The average of all 5 metrics are shown in Figure 21.

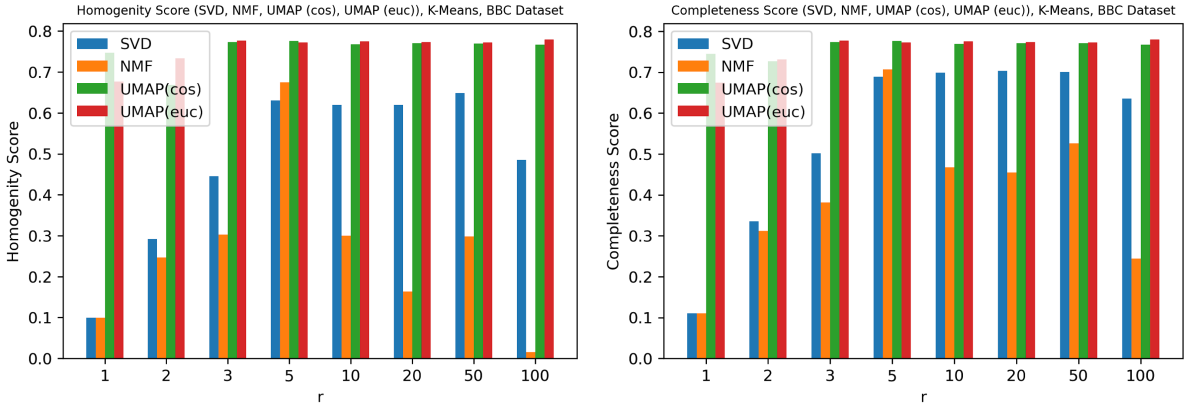


Figure 16 and 17: Plots of homogeneity and completeness for various values of r for both SVD, NMF and two forms of UMAP.

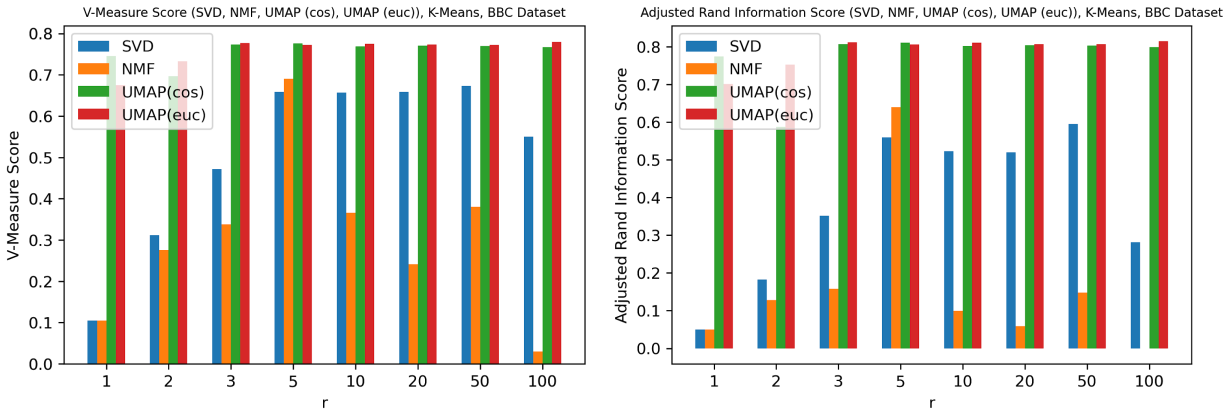


Figure 18 and 19: Plots of V-measure and ARI for various values of *r* for both SVD, NMF and two forms of UMAP.

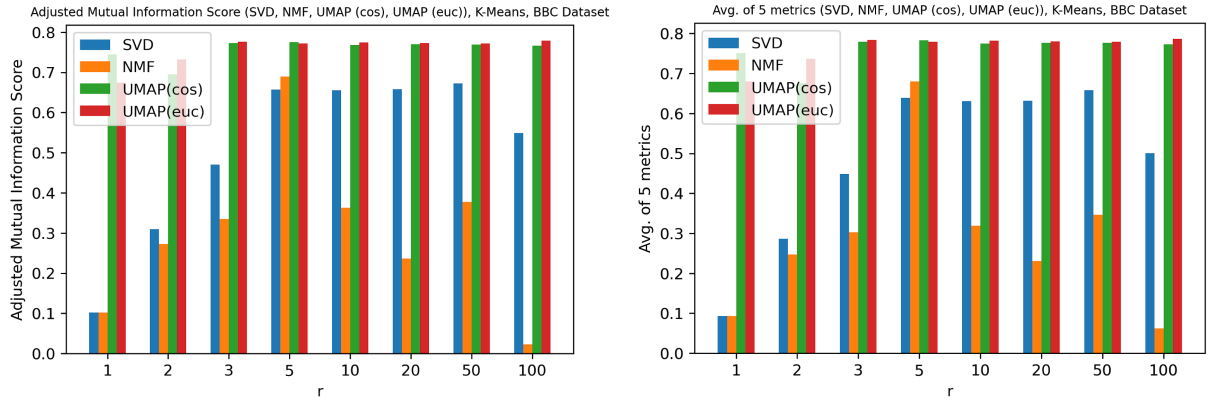


Figure 20 and 21: Plots of AMI and average of all 5 metrics for various values of *r* for both SVD, NMF and two forms of UMAP.

UMAP vs PCA (SVD and NMF): From Figures 16 and 17, we can see that UMAP (cosine) and UMAP (euclidean) performed consistently better for all values of ***r*** in every metric over SVD and NMF. This is because both SVD and NMF (which are simply principal component analysis (PCA) techniques) are least squares procedures and highly susceptible to outliers and noise compared to UMAP. Moreover, both SVD and NMF are linear projections of the high dimensional data in the directions that maximize the variance of the dataset, but are unable to capture nonlinear dependencies within the data. UMAP, on the other hand, attempts to make sure that the semantic and non-linear information, as well as global structure of the data is preserved during dimension reduction such that similar points in the embeddings are clustered together and dissimilar points are kept far apart, which aids the clustering algorithm innately. This is referred to as manifold learning. In fact, since UMAP focuses on preserving the information entropy regardless of the number of dimensions of the low-dimensional graph, the performance of UMAP is much more consistent across a wide range of ***r*** as shown in Figures 16 to 21 compared to SVD or NMF.

SVD vs NMF: As discussed in earlier questions, we see that the performance of NMF is much more erratic than that of the SVD. This is because NMF only allows positive entries in the reduced-rank feature matrix while SVD has no such restriction. As a result, SVD is able to better represent the higher-dimensional feature matrix,

providing a deeper factorization with lower information loss than NMF and causing the decomposition depth of NMF diminishes in high-dimensions (when ***r*** is large). Furthermore, SVD is much more deterministic than NMF, with a hierarchical and geometric basis ordered by relevance. NMF, on the other hand, does not consider the geometry in the feature space basis, which is essential for clustering in high dimensions. Since SVD produces a feature matrix with the most relevant features higher in the hierarchy, increasing the value of ***r*** does not cause significant drop or rise in clustering performance. From the K-means perspective, a feature matrix produced by SVD with higher values of ***r*** is not adding any significant distinguishable information, neither subtracting any important semantic information (due to noise) thanks to the ordering of the features. Last but not least, the results from SVD are unique while NMF is non-unique and stochastic, with no guarantees of convergence to the optimal feature matrix each time the function is called.

Explaining K-means performance in general: From Figures 16 to 21, we see that as ***r*** increases in general, the clustering metrics do not increase uniformly, especially for SVD and NMF. If ***r*** is more, then the amount of semantic and complex information and structure in the data available for clustering increases. Theoretically, as more information is preserved, it should lead to better clustering in terms of separation and compactness. However, a larger value of ***r*** indicates a high-dimensional and noisy feature matrix, which degrades K-means clustering performance. This is because in higher dimensions, the rapid increase in volume causes the data to become sparse in each dimension, causing the Euclidean distances to converge to a constant value between all sample points. Since all feature points become equidistant and the inertia is not normalized, the clustering algorithm cannot find centroids with sufficient distance among them, leading to poor clustering. This is very evident for the metrics related to NMF, which initially increase but then drop rapidly. Fortunately, since UMAP is able to preserve the meaning of the data while also removing outliers and noise in the hierarchical tree, K-means performs very similarly for both high and low values of ***r*** for UMAP.

UMAP (euclidean) vs. UMAP (cosine): Theoretically, UMAP with cosine distance metric should perform much better clustering than UMAP with Euclidean distance. This is because cosine similarity is not affected by the magnitude of the vectors, meaning that the length of the documents does not affect the distance metric, but rather it associates clusters based on angle between sample points. However, we see that both distance metrics perform similar. This indicates that all of the documents overall are of similar length.

The best value of ***r*** for the 4 dimension reduction techniques using K-means algorithm, as well as the 5 clustering metrics are as follows, according to Figure 21:

Best value of *r* for Euclidean UMAP (according to avg. metric): 100

Homogeneity (UMAP (euclidean), best *r*): 0.7792623218537925
Completeness (UMAP (euclidean), best *r*): 0.7797694659206051
V-measure (UMAP (euclidean), best *r*): 0.7795158114016759
Adjusted Rand-Index (UMAP (euclidean), best *r*): 0.8144656785590599

Adjusted Mutual Information Score (UMAP (euclidean), best *r*): 0.7787711893384506

Best value of *r* for Cosine UMAP (according to avg. metric): 5

Homogeneity (UMAP (cosine), best *r*): 0.7759972871042146
Completeness (UMAP (cosine), best *r*): 0.7762459012114491
V-measure (UMAP (cosine), best *r*): 0.7761215742482664
Adjusted Rand-Index (UMAP (cosine), best *r*): 0.8107291656585696
Adjusted Mutual Information Score (UMAP (cosine), best *r*): 0.7753656175306356

Best value of *r* for NMF (according to avg. metric): 5

Homogeneity (NMF, best *r*): 0.6749528460900887
Completeness (NMF, best *r*): 0.7070646037111105
V-measure (NMF, best *r*): 0.6906356597928979
Adjusted Rand-Index (NMF, best *r*): 0.6400335068305302
Adjusted Mutual Information Score (NMF, best *r*): 0.6895659419632774

Best value of *r* for SVD (according to avg. metric): 50

Homogeneity (SVD, best *r*): 0.6482818338095657
Completeness (SVD, best *r*): 0.7009056794330034
V-measure (SVD, best *r*): 0.6735674837344477
Adjusted Rand-Index (SVD, best *r*): 0.5946722914434844
Adjusted Mutual Information Score (SVD, best *r*): 0.672420502845868

Figures 22, 23, 24 and 25 show the contingency matrices for K-means clustering for the best value of ***r*** for each dimension reduction technique.

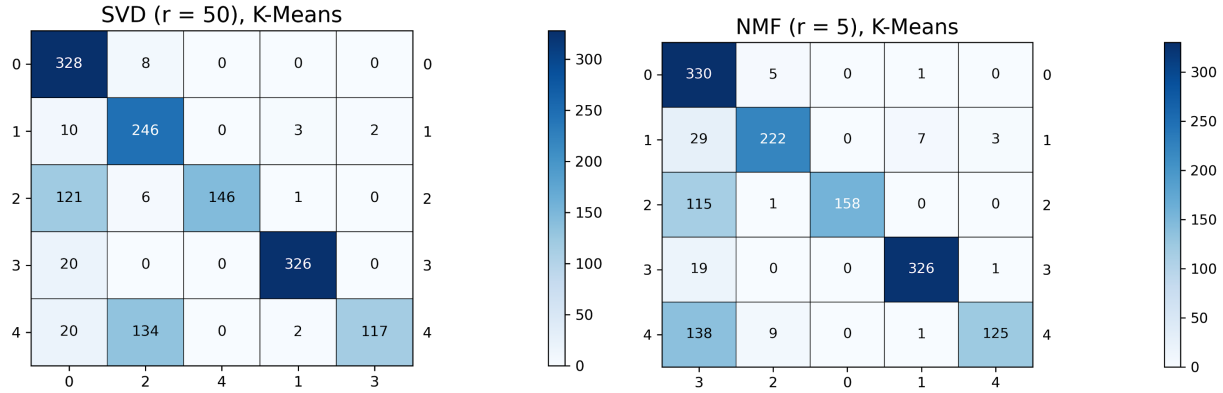


Figure 22 and 23: Contingency matrix visualization for K-means clustering with SVD (***r*** = 50) and NMF (***r*** = 5).

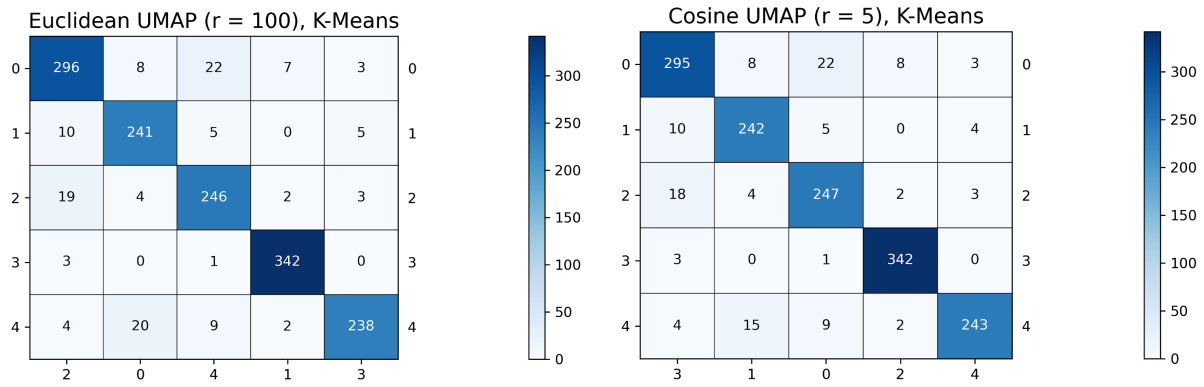


Figure 24 and 25: Contingency matrix visualization for K-means clustering with Euc. UMAP (***r*** = 100) and Cosine UMAP (***r*** = 5).

Judging from the contingency matrices in Figure 22 to 25, we can see that the UMAP matrices have a denser diagonal than SVD or NMF matrices, which means that UMAP is doing a better job at properly clustering the 5 class labels over PCA. The three problematic classes are 0, 3 and 4, which when examined was shown to be business, sports and entertainment. This is probably because these categories share similar vocabulary of words in their posts.

We chose to apply agglomerative clustering, DBSCAN and HDBSCAN on UMAP (cosine) dimensionality reduced feature matrix, with ***r*** = 10 (as both 5 and 10 performed similarly). The 5 clustering metrics for agglomerative clustering are as follows:

Agglomerative Clustering, Ward - Homogeneity: 0.764
Agglomerative Clustering, Ward - Completeness: 0.768
Agglomerative Clustering, Ward - V-measure: 0.766
Agglomerative Clustering, Ward - Adjusted Rand-Index: 0.793
Agglomerative Clustering, Ward - Adjusted Mutual Information Score: 0.765

Agglomerative Clustering, Single - Homogeneity: 0.316
Agglomerative Clustering, Single - Completeness: 0.633
Agglomerative Clustering, Single - V-measure: 0.422
Agglomerative Clustering, Single - Adjusted Rand-Index: 0.168
Agglomerative Clustering, Single - Adjusted Mutual Information Score: 0.419

Ward linkage vs Single linkage criteria: From the above metrics, we see that single linkage criteria performs significantly worse than ward linkage criteria. The single linkage criterion (nearest neighbour) defines the distance as the minimum distance between clusters data points, while the ward linkage criterion (minimal increase of sum-of-squares method) minimizes the sum of squared differences within all clusters. As a result, although single linkage is fast, efficient and works well on non-elliptical data, single linkage clustering is not robust to noisy data or outliers and leads to formation of long snakelike chains (as agglomerative clustering

follows “rich get richer” strategy), which causes data points to be erroneously clustered in one group. Ward linkage criterion (equivalent to K-means clustering), on the other hand, encourages formation of spherically tightly bound clusters and works well in presence of noise. This is also evident in the contingency matrices, shown in Figures 26 and 27. We see that single linkage agglomerative clustering has erroneously grouped most of the classes into one cluster (except class 3) via chaining.

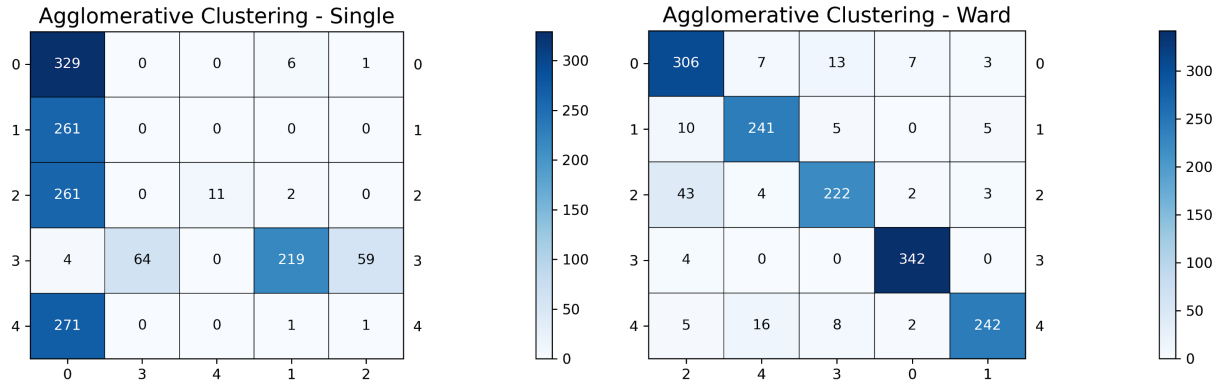


Figure 26 and 27: Contingency matrix visualization for agglomerative clustering (single vs. ward linkage) with UMAP (cosine).

After performing hyperparameter tuning of ***e*** and ***m***, the best values of ***e*** and ***m*** for DBSCAN and HDBSCAN and the associated clustering metrics are as follows:

Best value of epsilon and minimum number of samples hyperparameters for DBSCAN: 0.7 and 30 respectively

Homogeneity (DBSCAN, best hyperparameters): 0.7812601834858268
Completeness (DBSCAN, best hyperparameters): 0.6508983166825818
V-measure (DBSCAN, best hyperparameters): 0.7101461720364782
Adjusted Rand-Index (DBSCAN, best hyperparameters): 0.6991186381134069
Adjusted Mutual Information Score (DBSCAN, best hyperparameters): 0.708577013237118

Best value of epsilon and minimum number of samples hyperparameters for HDBSCAN: 0.01 and 15

Homogeneity (HDBSCAN, best hyperparameters): 0.5631797605582871
Completeness (HDBSCAN, best hyperparameters): 0.8104923940615928
V-measure (HDBSCAN, best hyperparameters): 0.6645732912133311
Adjusted Rand-Index (HDBSCAN, best hyperparameters): 0.5680108102696699
Adjusted Mutual Information Score (HDBSCAN, best hyperparameters): 0.6635260975877341

The contingency matrices for the best DBSCAN and HDBSCAN model are shown in Figure 28 and 29.

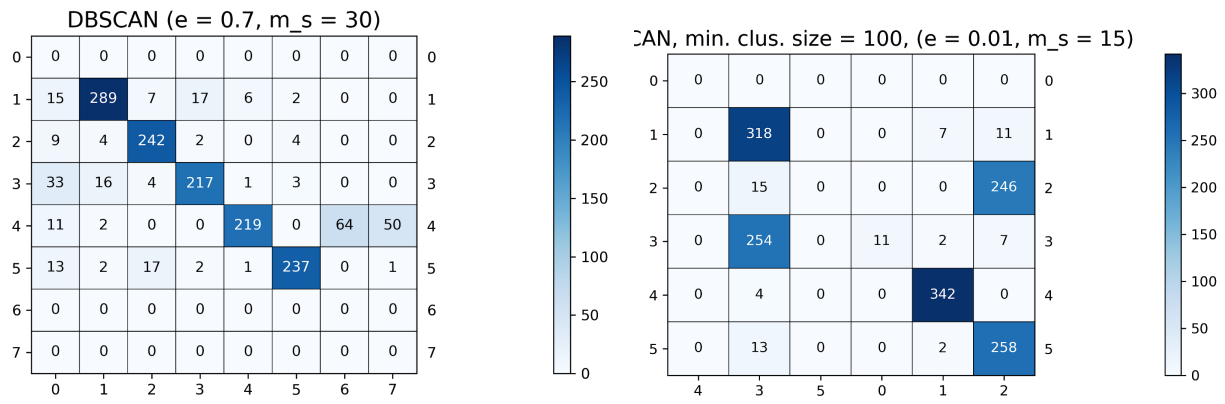


Figure 28 and 29: Contingency matrix visualization for DBSCAN (e = 0.7 and m = 30) and HDBSCAN (e = 0.01 and m = 15)

HDBSCAN vs DBSCAN: From the contingency matrix plots in Figure 28 and 29, we see that DBSCAN has successfully identified 5 major clusters (total 8) with strong diagonalization of the contingency plot within the data without any prior heuristics on the possible number of ground truth labels, while labelling some of the data as outliers. HDBSCAN, on the other hand, found only 3 major clusters (total 4). Judging from the metrics and contingency matrices, we see that DBSCAN performed better than HDBSCAN. This is probably due to the loss of clusters of low density for HDBSCAN, causing them to merge into other clusters, as evident from Figure 29 where multiple clusters are overlapping. A possible solution would be to treat the minimum cluster size as hyperparameter as well, possibly with the trade-off of formation of microclusters in larger clusters.

In summary the rank of all the clustering algorithms for the BBC dataset are as follows, in terms of the average of 5 clustering algorithms:

1. K-Means (5 clusters) with Euclidean UMAP (***r*** = 100) - 0.7863
2. K-Means (5 clusters) with Cosine UMAP (***r*** = 5) - 0.783
3. **Agglomerative clustering (ward linkage criterion) with Cosine UMAP (*r* = 10) - 0.7712 (our personal recommendation)**
4. DBSCAN (***e*** = 0.7 and ***m*** = 30) with Cosine UMAP (***r*** = 10) - 0.7100
5. K-Means (5 clusters) with NMF (***r*** = 5) - 0.6804
6. K-Means (5 clusters) with SVD (***r*** = 50) - 0.6580
7. HDBSCAN (***e*** = 0.01 and ***m*** = 15) with Cosine UMAP (***r*** = 10) - 0.6540
8. Agglomerative clustering (single linkage criterion) with Cosine UMAP - 0.3916

In our opinion, we would recommend using agglomerative clustering (ward linkage) with Cosine UMAP feature matrix over K-Means clustering due to the following reasons:

- Agglomerative clustering has no initialization sensitivity issue, yielding reproducible results with fewer assumptions about the data distribution. K-Means on the other hand, is dependent on the initial

position and centroids as well as pre-specified value of K, which affects the final outcome. In addition, K-means also assumes the groups of data are well separated and spherical.

- Agglomerative clustering can handle any forms of similarity or distance, but K-means strictly uses Euclidean distance in the inertia loss function, which does not scale well in higher dimensions as discussed multiple times in previous questions.
- Agglomerative clustering is hierarchical and thus able to coarsely or finely group the data (nested clusters arranged as trees), whereas k-means simply partitions the data into non-overlapping subsets.

project_219_2_saha_young

January 27, 2021

[1]:
```
import warnings
warnings.filterwarnings('ignore')
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer,␣
↪TfidfVectorizer
from sklearn.cluster import KMeans,  AgglomerativeClustering, DBSCAN
from sklearn.metrics.cluster import contingency_matrix, homogeneity_score,␣
↪completeness_score, adjusted_rand_score, adjusted_mutual_info_score,␣
↪v_measure_score
from sklearn.metrics import confusion_matrix
from scipy.optimize import linear_sum_assignment
from sklearn.decomposition import TruncatedSVD, NMF
from plotmat import plot_mat
from matplotlib import pyplot as plt
import numpy as np
import random
import pandas as pd
import umap.umap_ as umap #0.4.6
import hdbscan
import joblib #0.17.0
np.random.seed(0)
random.seed(0)
```

0.1 Question 1

[2]:
```
categories = ['comp.graphics','comp.os.ms-windows.misc','comp.sys.ibm.pc.
↪hardware',
              'comp.sys.mac.hardware','rec.autos','rec.motorcycles','rec.sport.
↪baseball','rec.sport.hockey']
dataset = fetch_20newsgroups(subset = 'all', categories = categories,shuffle =␣
↪True, random_state = 0,remove=('headers','footers'))
vectorizer = CountVectorizer(stop_words='english',min_df=3)
tfidf_transformer = TfidfTransformer()
data_feat_vec = vectorizer.fit_transform(dataset.data)
data_feat = tfidf_transformer.fit_transform(data_feat_vec)

y_GT = []
```

```
for label in dataset.target:
    if label < 4:
        y_GT.append(0)
    else:
        y_GT.append(1)

print('Shape if TF-IDF matrix: ',data_feat.shape)
```

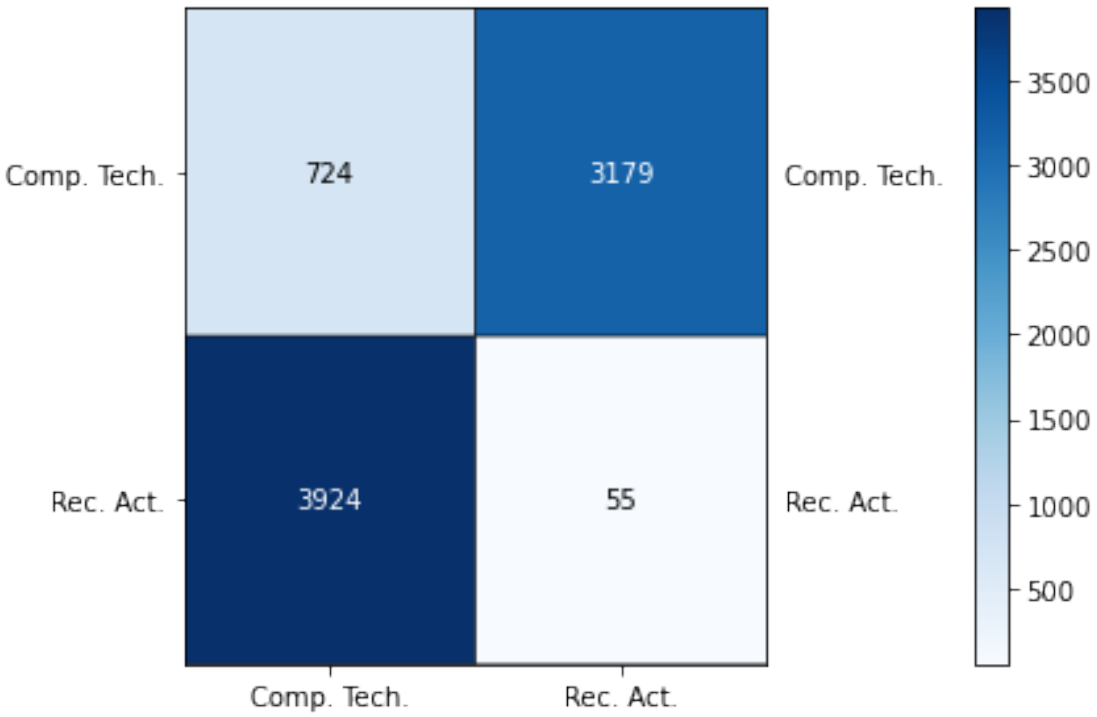
```
Shape if TF-IDF matrix:  (7882, 23522)
```

0.2 Question 2

[3]:
```
km = KMeans(n_clusters=2, init='k-means++', max_iter=1000, n_init=30,␣
↪random_state=0,n_jobs=-1)
km.fit(data_feat)
```

[3]:
```
KMeans(max_iter=1000, n_clusters=2, n_init=30, n_jobs=-1, random_state=0)
```

[4]:
```
plot_mat(contingency_matrix(y_GT,km.labels_),size=(6,4),xticklabels = ['Comp.␣
↪Tech.','Rec. Act.'],yticklabels = ['Comp. Tech.','Rec. Act.'],pic_fname =␣
↪'Q2.png')
```



0.3 Question 3

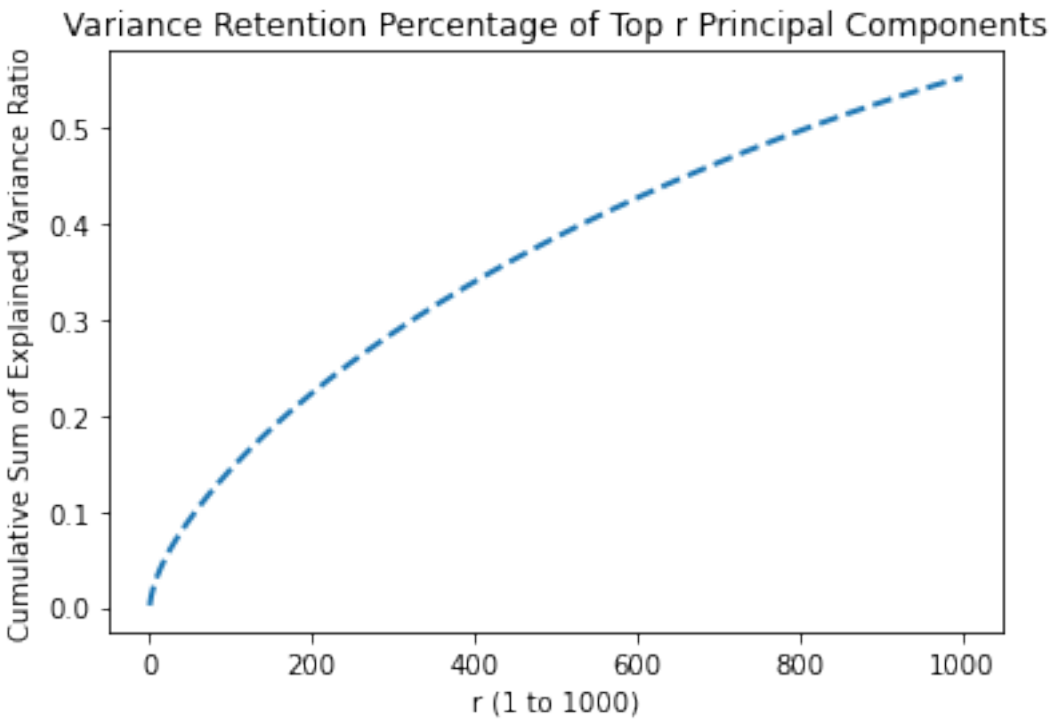
[5]:
```
print("Homogeneity: %0.3f" % homogeneity_score(y_GT, km.labels_))
print("Completeness: %0.3f" % completeness_score(y_GT, km.labels_))
print("V-measure: %0.3f" % v_measure_score(y_GT, km.labels_))
print("Adjusted Rand-Index: %.3f"% adjusted_rand_score(y_GT, km.labels_))
print("Adjusted Mutual Information Score: %.3f"%␣
 ↪adjusted_mutual_info_score(y_GT, km.labels_))
```

```
Homogeneity: 0.581
Completeness: 0.595
V-measure: 0.588
Adjusted Rand-Index: 0.644
Adjusted Mutual Information Score: 0.588
```

0.4 Question 4

[6]:
```
svd = TruncatedSVD(n_components=1000, random_state=0)
LSI = svd.fit_transform(data_feat)
```

[7]:
```
plt.plot(np.linspace(1,1000,1000),np.cumsum(svd.
 ↪explained_variance_ratio_),lw=2,linestyle='--')
plt.title('Variance Retention Percentage of Top r Principal Components')
plt.ylabel('Cumulative Sum of Explained Variance Ratio')
plt.xlabel('r (1 to 1000)')
plt.savefig('Q4.png',dpi=300,bbox_inches='tight')
plt.show()
```



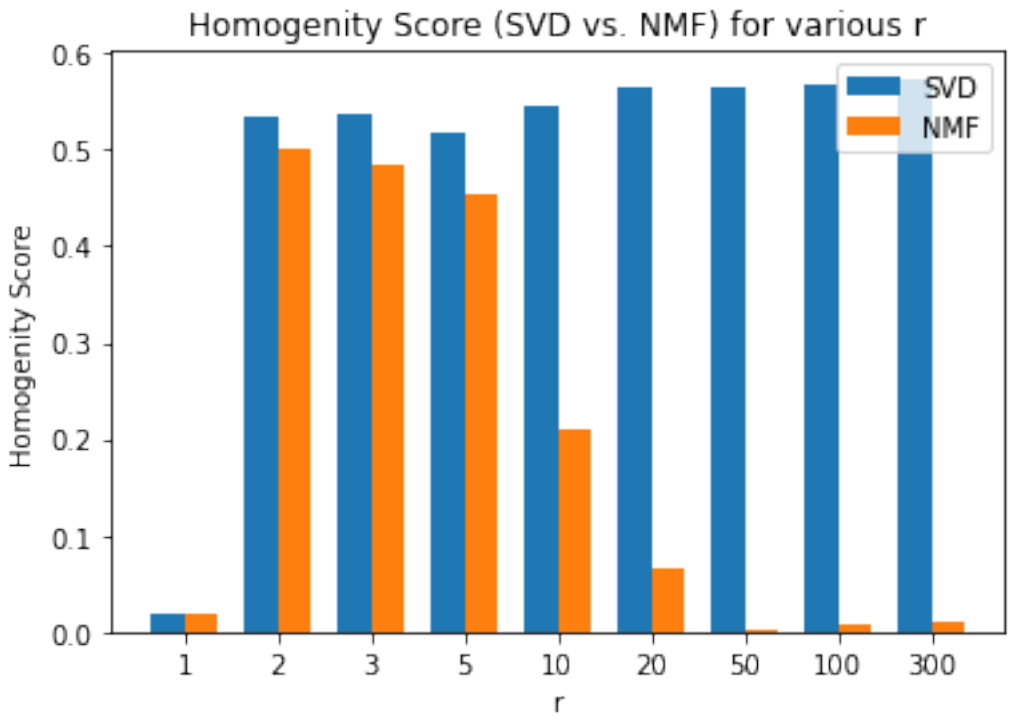
0.5 Question 5

[8]:
```
svd_hs = []
svd_cs = []
svd_vs = []
svd_ari = []
svd_ms = []
nmf_hs = []
nmf_cs = []
nmf_vs = []
nmf_ari = []
nmf_ms = []

r = [1,2,3,5,10,20,50,100,300]

for i in range(len(r)):
    print('Testing SVD for r = ',r[i])
    svd = TruncatedSVD(n_components=r[i], random_state=0)
    svd_km = svd.fit_transform(data_feat)
    kmean_svd = km.fit(svd_km)
    svd_hs.append(homogeneity_score(y_GT, kmean_svd.labels_))
    svd_cs.append(completeness_score(y_GT, kmean_svd.labels_))
    svd_vs.append(v_measure_score(y_GT, kmean_svd.labels_))
```

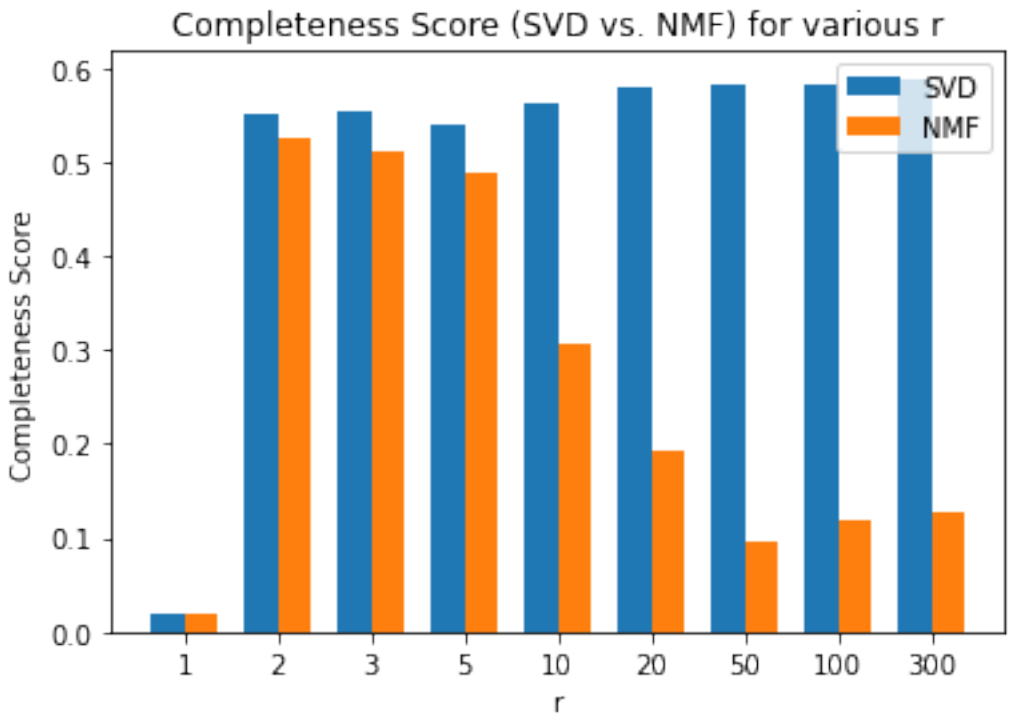
```
    svd_ari.append(adjusted_rand_score(y_GT, kmean_svd.labels_))
    svd_ms.append(adjusted_mutual_info_score(y_GT, kmean_svd.labels_))
    print('Testing NMF for r = ',r[i])
    nmf = NMF(n_components=r[i], init='random', random_state=0, max_iter=400)
    nmf_km = nmf.fit_transform(data_feat)
    kmean_nmf = km.fit(nmf_km)
    nmf_hs.append(homogeneity_score(y_GT, kmean_nmf.labels_))
    nmf_cs.append(completeness_score(y_GT, kmean_nmf.labels_))
    nmf_vs.append(v_measure_score(y_GT, kmean_nmf.labels_))
    nmf_ari.append(adjusted_rand_score(y_GT, kmean_nmf.labels_))
    nmf_ms.append(adjusted_mutual_info_score(y_GT, kmean_nmf.labels_))

print('Done testing.')
```

```
Testing SVD for r =  1
Testing NMF for r =  1
Testing SVD for r =  2
Testing NMF for r =  2
Testing SVD for r =  3
Testing NMF for r =  3
Testing SVD for r =  5
Testing NMF for r =  5
Testing SVD for r =  10
Testing NMF for r =  10
Testing SVD for r =  20
Testing NMF for r =  20
Testing SVD for r =  50
Testing NMF for r =  50
Testing SVD for r =  100
Testing NMF for r =  100
Testing SVD for r =  300
Testing NMF for r =  300
Done testing.
```

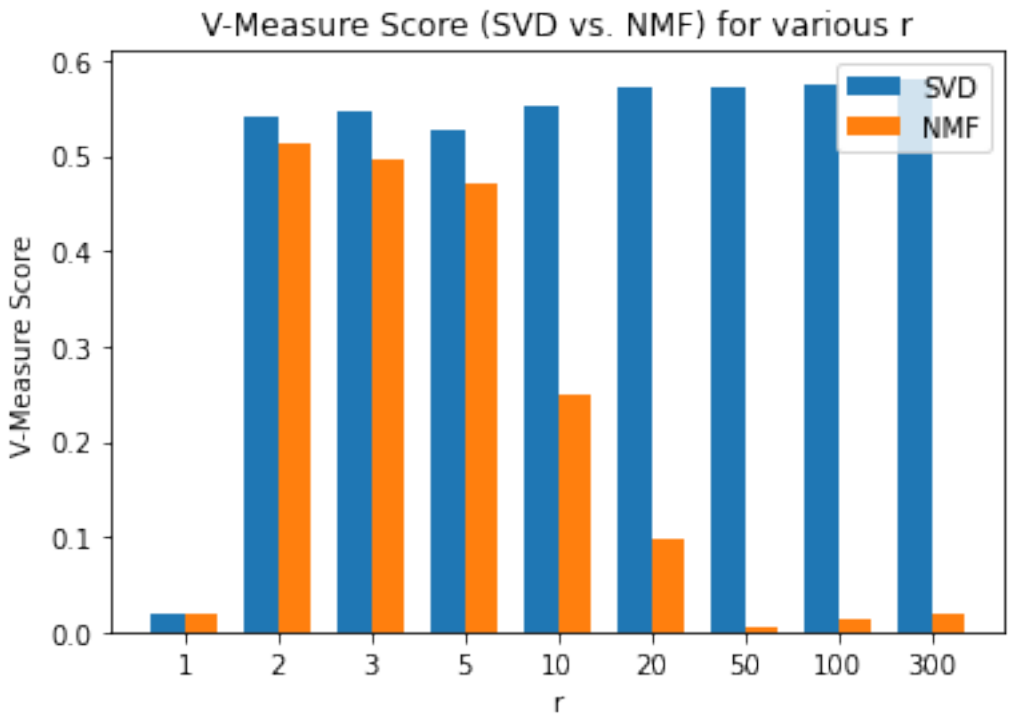
[9]:
```
width = 0.35
fig, ax = plt.subplots()
rects1 = ax.bar(np.arange(len(r)) - width/2, svd_hs, width, label='SVD')
rects2 = ax.bar(np.arange(len(r)) + width/2, nmf_hs, width, label='NMF')
ax.set_ylabel('Homogenity Score')
ax.set_title('Homogenity Score (SVD vs. NMF) for various r')
ax.set_xticks(np.arange(len(r)))
ax.set_xticklabels(r)
ax.set_xlabel('r')
ax.legend()
plt.savefig('Q51.png',dpi=300,bbox_inches='tight')
plt.show()
```



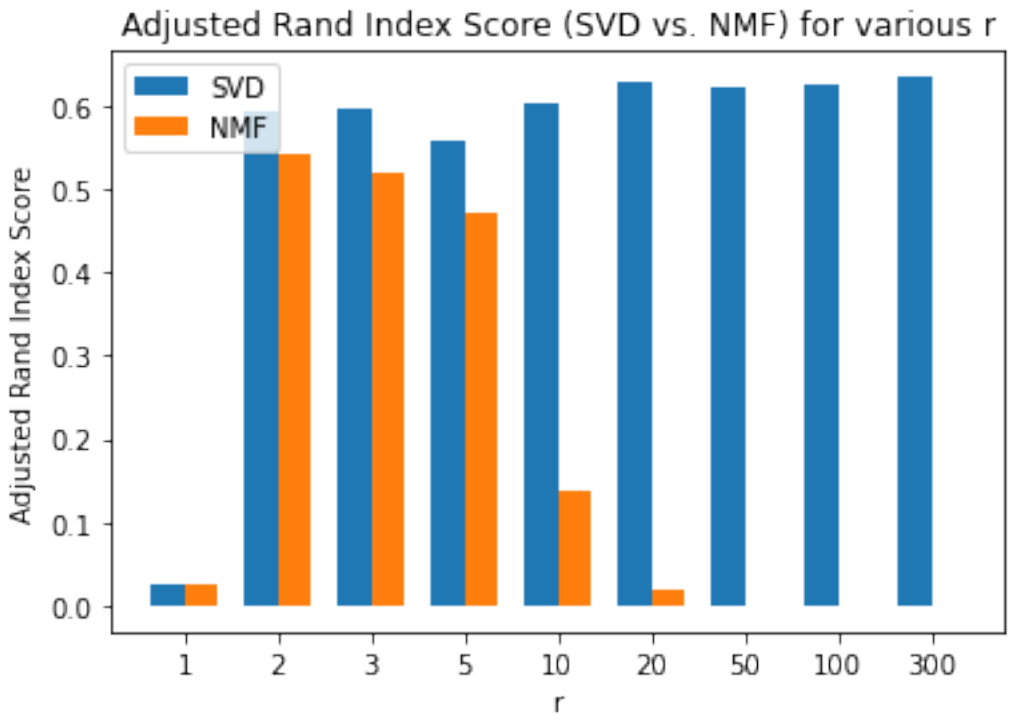
[10]:
```
fig, ax = plt.subplots()
rects1 = ax.bar(np.arange(len(r)) - width/2, svd_cs, width, label='SVD')
rects2 = ax.bar(np.arange(len(r)) + width/2, nmf_cs, width, label='NMF')
ax.set_ylabel('Completeness Score')
ax.set_title('Completeness Score (SVD vs. NMF) for various r')
ax.set_xticks(np.arange(len(r)))
ax.set_xticklabels(r)
ax.set_xlabel('r')
ax.legend()
plt.savefig('Q52.png',dpi=300,bbox_inches='tight')
plt.show()
```



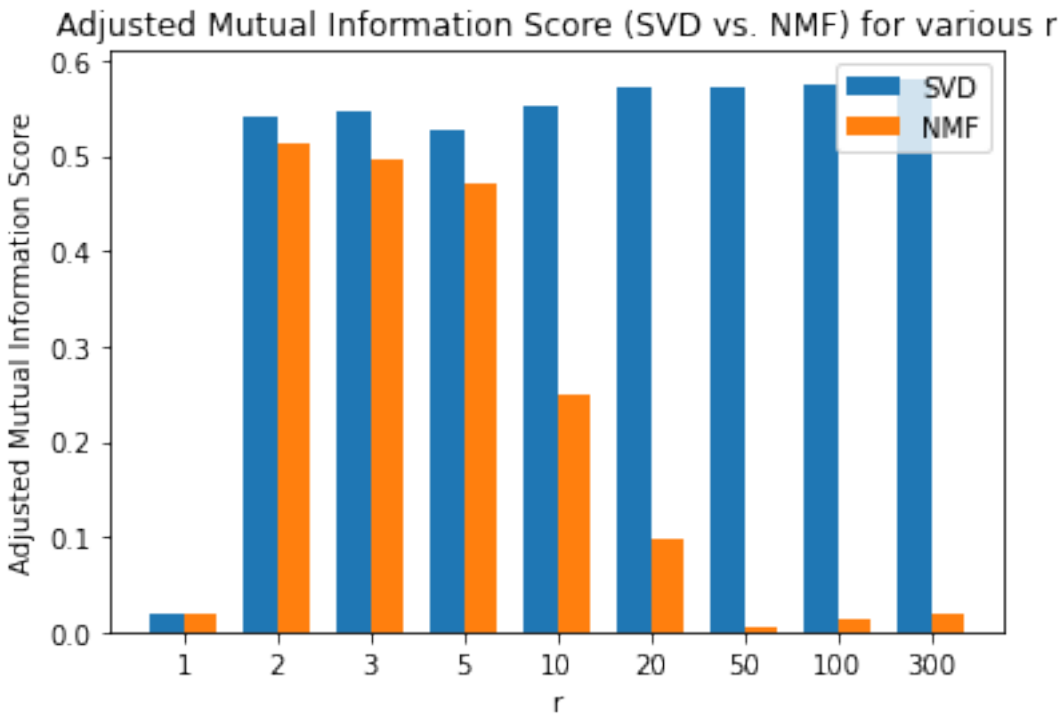
[11]:
```
fig, ax = plt.subplots()
rects1 = ax.bar(np.arange(len(r)) - width/2, svd_vs, width, label='SVD')
rects2 = ax.bar(np.arange(len(r)) + width/2, nmf_vs, width, label='NMF')
ax.set_ylabel('V-Measure Score')
ax.set_title('V-Measure Score (SVD vs. NMF) for various r')
ax.set_xticks(np.arange(len(r)))
ax.set_xticklabels(r)
ax.set_xlabel('r')
ax.legend()
plt.savefig('Q53.png',dpi=300,bbox_inches='tight')
plt.show()
```



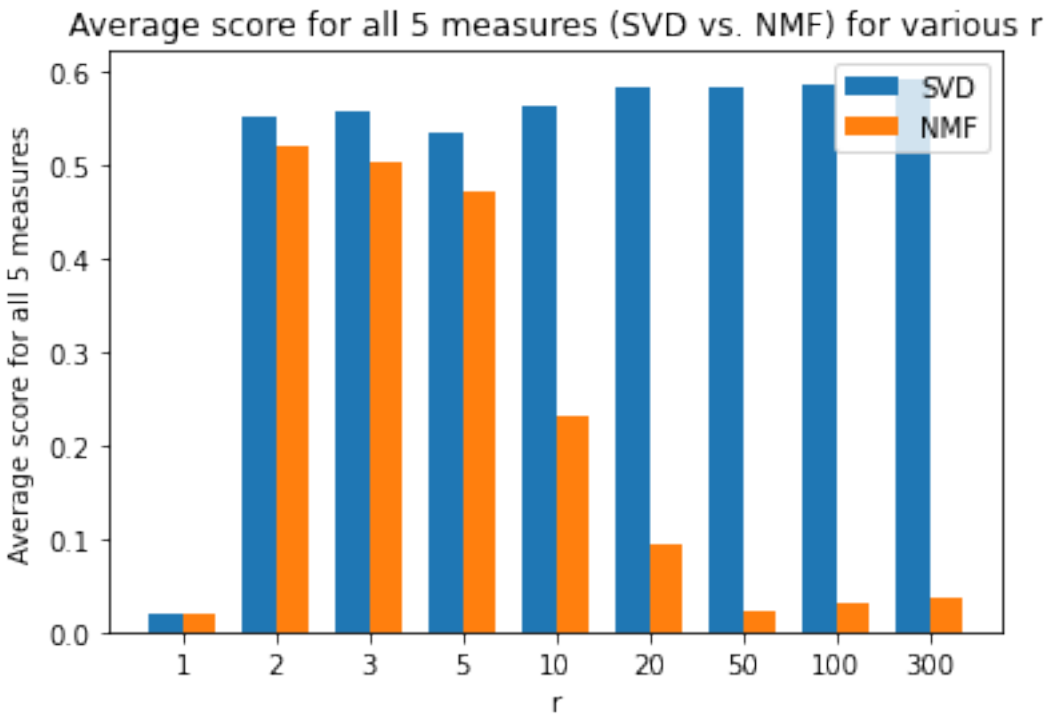
[12]:
```
fig, ax = plt.subplots()
rects1 = ax.bar(np.arange(len(r)) - width/2, svd_ari, width, label='SVD')
rects2 = ax.bar(np.arange(len(r)) + width/2, nmf_ari, width, label='NMF')
ax.set_ylabel('Adjusted Rand Index Score')
ax.set_title('Adjusted Rand Index Score (SVD vs. NMF) for various r')
ax.set_xticks(np.arange(len(r)))
ax.set_xticklabels(r)
ax.set_xlabel('r')
ax.legend()
plt.savefig('Q54.png',dpi=300,bbox_inches='tight')
plt.show()
```



[13]:
```
fig, ax = plt.subplots()
rects1 = ax.bar(np.arange(len(r)) - width/2, svd_ms, width, label='SVD')
rects2 = ax.bar(np.arange(len(r)) + width/2, nmf_ms, width, label='NMF')
ax.set_ylabel('Adjusted Mutual Information Score')
ax.set_title('Adjusted Mutual Information Score (SVD vs. NMF) for various r')
ax.set_xticks(np.arange(len(r)))
ax.set_xticklabels(r)
ax.set_xlabel('r')
ax.legend()
plt.savefig('Q55.png',dpi=300,bbox_inches='tight')
plt.show()
```



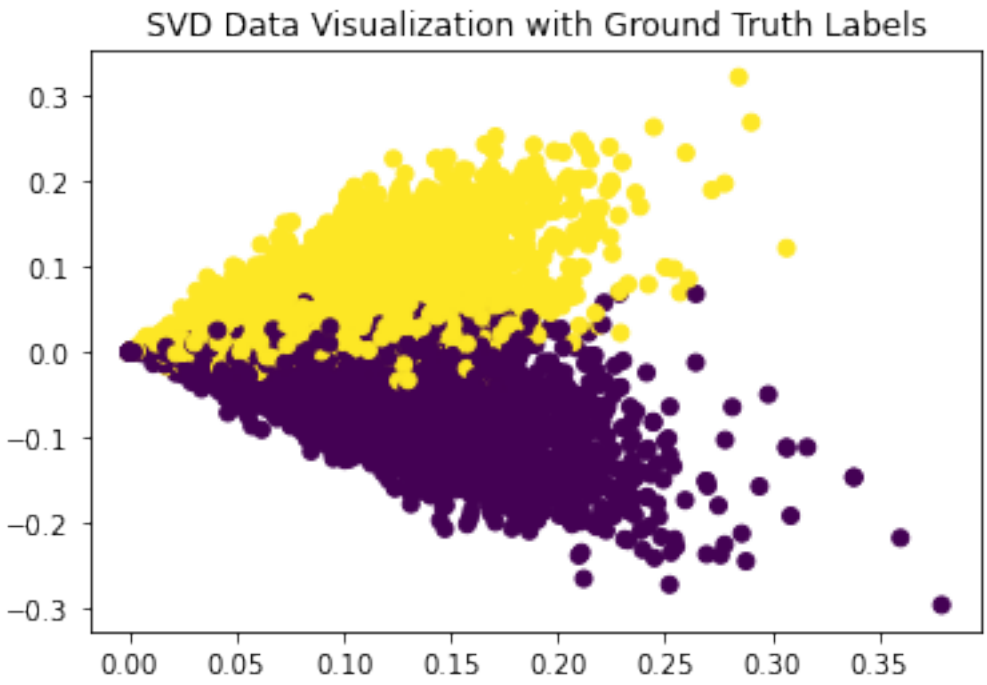
[14]:
```
fig, ax = plt.subplots()
rects1 = ax.bar(np.arange(len(r)) - width/2, [y/5 for y in [sum(x) for x in␣
 ↪zip(svd_hs, svd_cs, svd_vs, svd_ari, svd_ms)]], width, label='SVD')
rects2 = ax.bar(np.arange(len(r)) + width/2, [y/5 for y in [sum(x) for x in␣
 ↪zip(nmf_hs, nmf_cs, nmf_vs, nmf_ari, nmf_ms)]], width, label='NMF')
ax.set_ylabel('Average score for all 5 measures')
ax.set_title('Average score for all 5 measures (SVD vs. NMF) for various r')
ax.set_xticks(np.arange(len(r)))
ax.set_xticklabels(r)
ax.set_xlabel('r')
ax.legend()
plt.savefig('Q56.png',dpi=300,bbox_inches='tight')
plt.show()
```

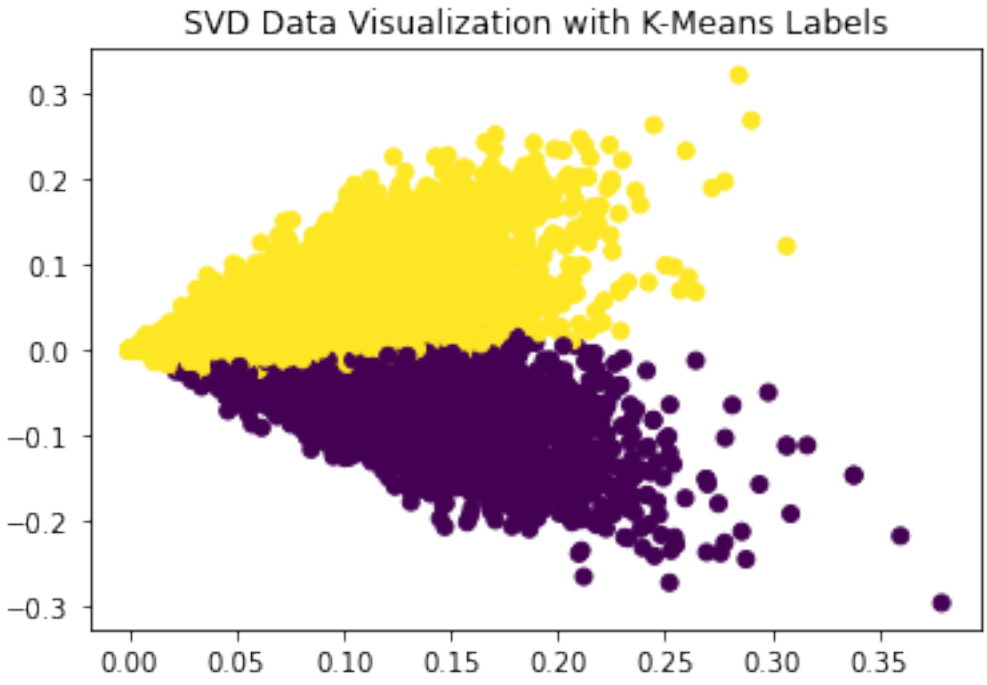


0.6 Question 7

[8]:
```
#best r for SVD: 20. Best r for NMF: 2
svd = TruncatedSVD(n_components=20, random_state=0)
svd_km = svd.fit_transform(data_feat)
y_svd = km.fit_predict(svd_km)
nmf = NMF(n_components=2, init='random', random_state=0, max_iter=400)
nmf_km = nmf.fit_transform(data_feat)
nmf_svd = km.fit_predict(nmf_km)
```

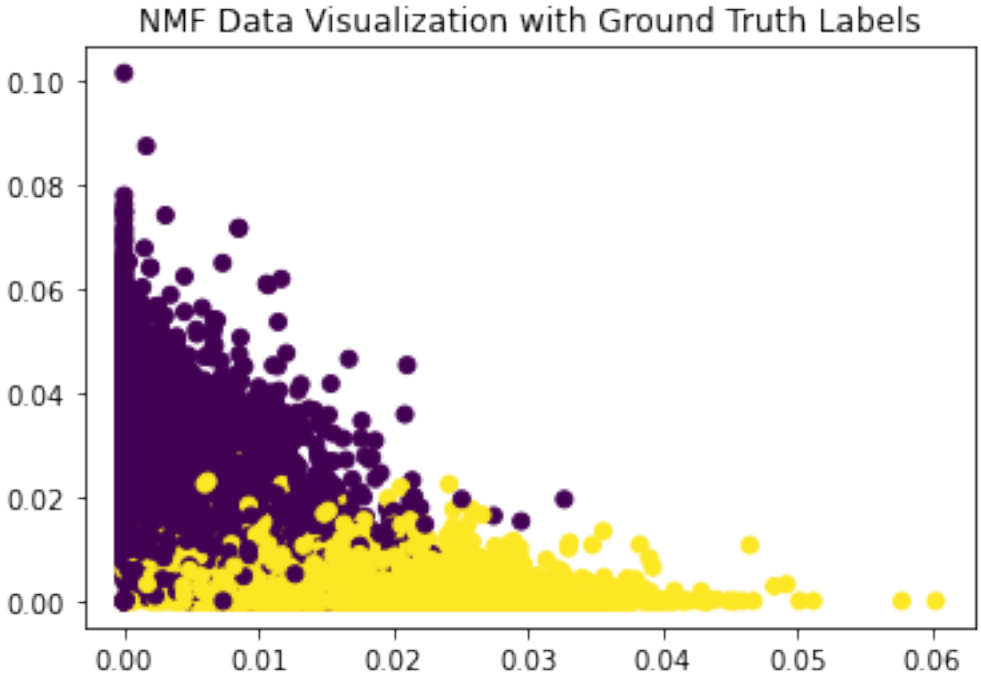
[9]:
```
plt.scatter(svd_km[:,0],svd_km[:,1],c=y_GT)
plt.title("SVD Data Visualization with Ground Truth Labels")
plt.savefig('Q71.png',dpi=300,bbox_inches='tight')
plt.show()
plt.scatter(svd_km[:,0],svd_km[:,1],c=y_svd)
plt.title("SVD Data Visualization with K-Means Labels")
plt.savefig('Q72.png',dpi=300,bbox_inches='tight')
plt.show()
```

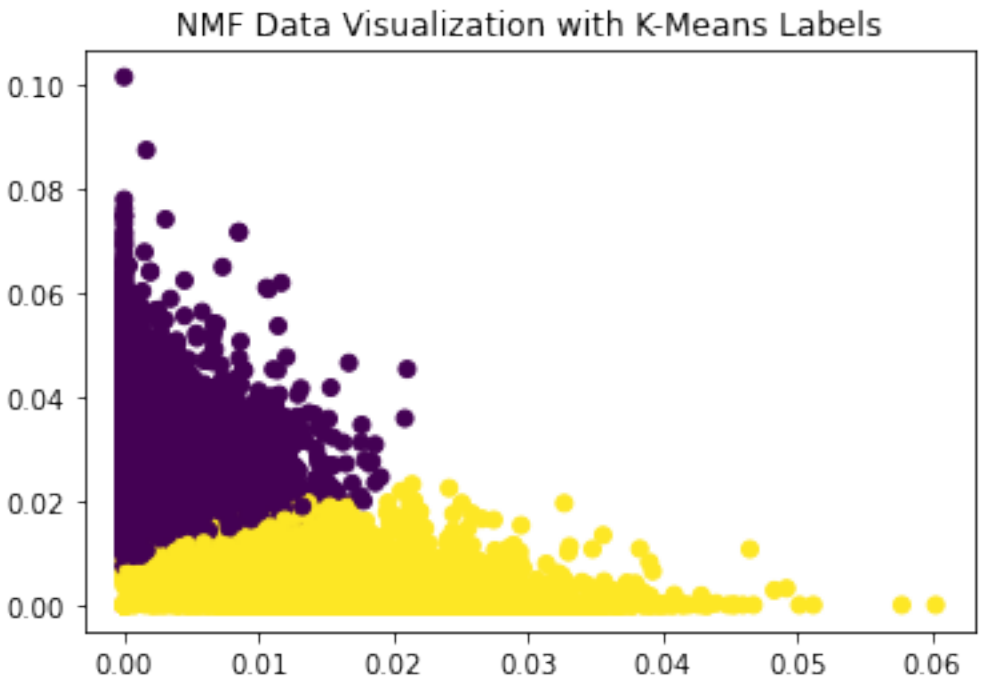




[10]:
```
plt.scatter(nmf_km[:,0],nmf_km[:,1],c=y_GT)
plt.title("NMF Data Visualization with Ground Truth Labels")
```

```
plt.savefig('Q73.png',dpi=300,bbox_inches='tight')
plt.show()
plt.scatter(nmf_km[:,0],nmf_km[:,1],c=y_svd)
plt.title("NMF Data Visualization with K-Means Labels")
plt.savefig('Q74.png',dpi=300,bbox_inches='tight')
plt.show()
```





0.7 Question 9

[11]:
```
dataset = fetch_20newsgroups(subset = 'all',shuffle = True, random_state =␣
↪0,remove=('headers','footers'))
vectorizer = CountVectorizer(stop_words='english',min_df=3)
tfidf_transformer = TfidfTransformer()
data_feat_vec = vectorizer.fit_transform(dataset.data)
data_feat = tfidf_transformer.fit_transform(data_feat_vec)
```

[19]:
```
svd_hs = []
svd_cs = []
svd_vs = []
svd_ari = []
svd_ms = []

km = KMeans(n_clusters=20, init='k-means++', max_iter=1000, n_init=30,␣
↪random_state=0,n_jobs=-1)

for i in range(len(r)):
    print('Testing SVD for r = ',r[i])
    svd = TruncatedSVD(n_components=r[i], random_state=0)
    svd_km = svd.fit_transform(data_feat)
    kmean_svd = km.fit(svd_km)
    svd_hs.append(homogeneity_score(dataset.target, kmean_svd.labels_))
```

```
    svd_cs.append(completeness_score(dataset.target, kmean_svd.labels_))
    svd_vs.append(v_measure_score(dataset.target, kmean_svd.labels_))
    svd_ari.append(adjusted_rand_score(dataset.target, kmean_svd.labels_))
    svd_ms.append(adjusted_mutual_info_score(dataset.target, kmean_svd.labels_))
print('Done testing')
```

```
Testing SVD for r =  1
Testing SVD for r =  2
Testing SVD for r =  3
Testing SVD for r =  5
Testing SVD for r =  10
Testing SVD for r =  20
Testing SVD for r =  50
Testing SVD for r =  100
Testing SVD for r =  300
Done testing
```

[20]:
```
print('R: ',r)
print('Homogeneity (SVD, various r): ',svd_hs)
print('Completeness (SVD, various r): ',svd_cs)
print('V-measure (SVD, various r): ',svd_vs)
print('Adjusted Rand-Index (SVD, various r): ',svd_ari)
print('Adjusted Mutual Information Score (SVD, various r): ',svd_ms)
```

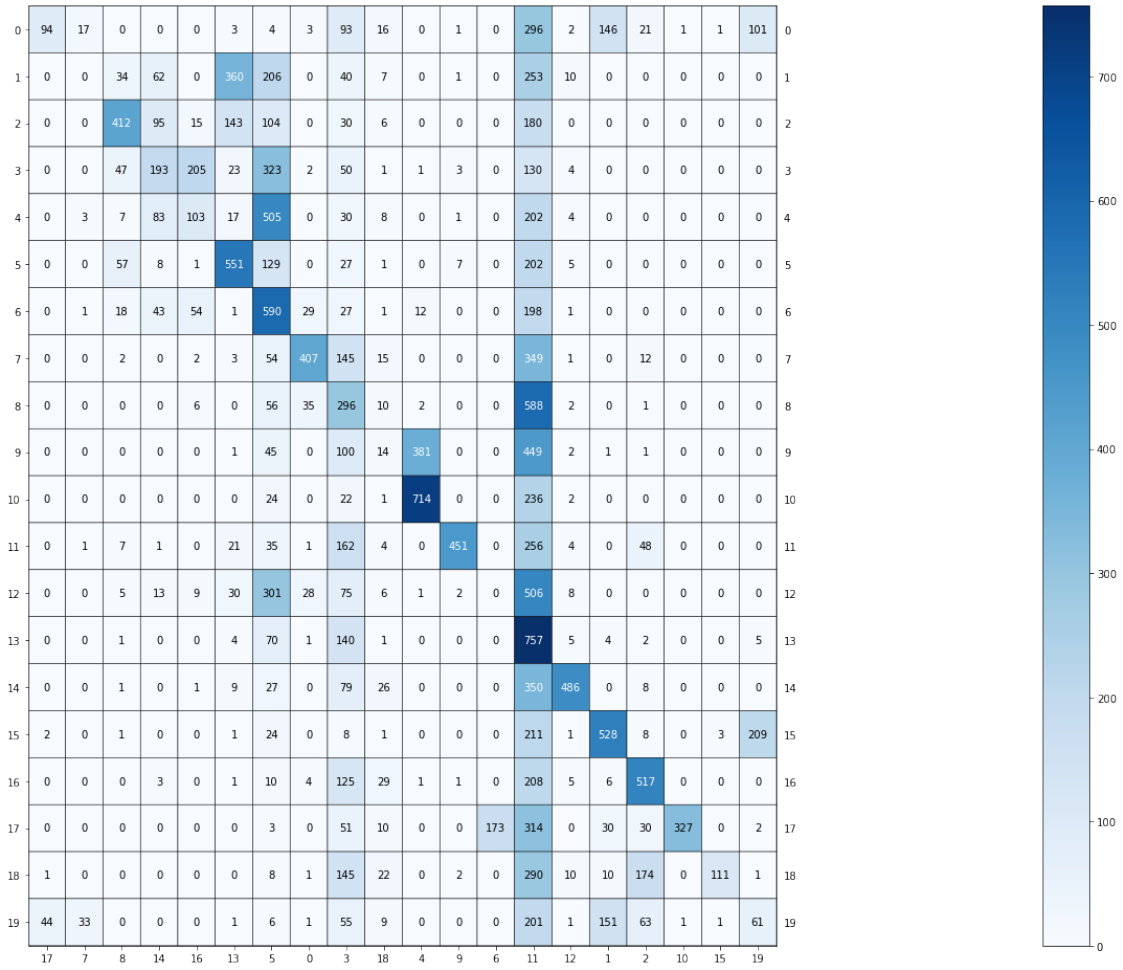
```
R:  [1, 2, 3, 5, 10, 20, 50, 100, 300]
Homogeneity (SVD, various r):  [0.024187148612053004, 0.2125950892676866,
0.24738131493939977, 0.3214333778771125, 0.3245499574754725, 0.333236617043135,
0.3458766189788096, 0.3209048193470938, 0.32102649871994504]
Completeness (SVD, various r):  [0.026467776667175487, 0.22476838950590647,
0.26593730647544095, 0.34925566861779905, 0.3538692799788343,
0.3755693336566379, 0.43446715637092265, 0.3832675865359189,
0.40179024944743597]
V-measure (SVD, various r):  [0.025276122475778838, 0.21851232739210352,
0.2563239197752381, 0.33476744519160806, 0.338576070454494, 0.3531388360645786,
0.38514315318415665, 0.3493247238641128, 0.3568963152194555]
Adjusted Rand-Index (SVD, various r):  [0.005242558241838782,
0.06556764766862869, 0.08392102022753462, 0.12656215193560497,
0.12266231553925155, 0.12053277796305269, 0.11042604980495854,
0.09564019615748767, 0.10405911055652439]
Adjusted Mutual Information Score (SVD, various r):  [0.021952877952663864,
0.21591333596299245, 0.2538214903064683, 0.3325230263533299, 0.3363424582136801,
0.35091614810495114, 0.3829063568650131, 0.347024461068902, 0.3545680944238939]
```

[21]:
```
avg_metrics = [y/5 for y in [sum(x) for x in zip(svd_hs, svd_cs, svd_vs,␣
 ↪svd_ari, svd_ms)]]
best_r_SVD = r[avg_metrics.index(max(avg_metrics))]
```

```
print('Best value of r for SVD (according to avg. metric): ', best_r_SVD, ',␣
 ↪avg. value of 5 metrics: ',max(avg_metrics))
```

```
Best value of r for SVD (according to avg. metric):  50 , avg. value of 5
metrics:  0.3317638670407721
```

[22]:
```
svd = TruncatedSVD(n_components=best_r_SVD, random_state=0)
svd_km = svd.fit_transform(data_feat)
kmean_svd = km.fit(svd_km)
cm = confusion_matrix(dataset.target, km.labels_)
rows, cols = linear_sum_assignment(cm, maximize=True)
plot_mat(cm[rows[:, np.newaxis], cols], xticklabels=cols, yticklabels=rows,␣
 ↪size=(15,13),pic_fname = 'Q9.png')
print("Homogeneity (SVD, best r): %0.3f" % homogeneity_score(dataset.target, km.
 ↪labels_))
print("Completeness (SVD, best r): %0.3f" % completeness_score(dataset.target,␣
 ↪km.labels_))
print("V-measure (SVD, best r): %0.3f" % v_measure_score(dataset.target, km.
 ↪labels_))
print("Adjusted Rand-Index (SVD, best r): %.3f"% adjusted_rand_score(dataset.
 ↪target, km.labels_))
print("Adjusted Mutual Information Score (SVD, best r): %.3f"%␣
 ↪adjusted_mutual_info_score(dataset.target, km.labels_))
```




```
Homogeneity (SVD, best r): 0.346
Completeness (SVD, best r): 0.434
V-measure (SVD, best r): 0.385
Adjusted Rand-Index (SVD, best r): 0.110
Adjusted Mutual Information Score (SVD, best r): 0.383
```

0.8 Question 10

[23]:
```
nmf_hs = []
nmf_cs = []
nmf_vs = []
nmf_ari = []
nmf_ms = []

for i in range(len(r)-1):
    print('Testing NMF for r = ',r[i])
    nmf = NMF(n_components=r[i], init='random', random_state=0, max_iter=400)
```

```
    nmf_km = nmf.fit_transform(data_feat)
    kmean_nmf = km.fit(nmf_km)
    nmf_hs.append(homogeneity_score(dataset.target, kmean_nmf.labels_))
    nmf_cs.append(completeness_score(dataset.target, kmean_nmf.labels_))
    nmf_vs.append(v_measure_score(dataset.target, kmean_nmf.labels_))
    nmf_ari.append(adjusted_rand_score(dataset.target, kmean_nmf.labels_))
    nmf_ms.append(adjusted_mutual_info_score(dataset.target, kmean_nmf.labels_))
print('Done testing')
```

```
Testing NMF for r =  1
Testing NMF for r =  2
Testing NMF for r =  3
Testing NMF for r =  5
Testing NMF for r =  10
Testing NMF for r =  20
Testing NMF for r =  50
Testing NMF for r =  100
Done testing
```

[24]:
```
print('R: ',r[:-1])
print('Homogeneity (NMF, various r): ',nmf_hs)
print('Completeness (NMF, various r): ',nmf_cs)
print('V-measure (NMF, various r): ',nmf_vs)
print('Adjusted Rand-Index (NMF, various r): ',nmf_ari)
print('Adjusted Mutual Information Score (NMF, various r): ',nmf_ms)
```

```
R:  [1, 2, 3, 5, 10, 20, 50, 100]
Homogeneity (NMF, various r):  [0.02418293013833349, 0.1910547790262287,
0.21967746007150765, 0.26683550314528043, 0.2888312487267663, 0.311480021698195,
0.25463244447819394, 0.14029330462291845]
Completeness (NMF, various r):  [0.02646303476881646, 0.20464240207954457,
0.2570744662468709, 0.28870335515719603, 0.3206130779777108, 0.3724086260503605,
0.34717245207687336, 0.19268911998670613]
V-measure (NMF, various r):  [0.025271656774071373, 0.19761530167814273,
0.236909229629993, 0.27733903355922124, 0.3038934702081099, 0.33923021621908855,
0.29378747376055575, 0.16236889042723177]
Adjusted Rand-Index (NMF, various r):  [0.0052417481099916625,
0.057633539892422626, 0.06731888280057871, 0.08697997264737248,
0.1018015613260908, 0.0956757455706522, 0.06671757556446743,
0.026550532092819917]
Adjusted Mutual Information Score (NMF, various r):  [0.021948404871629232,
0.19492006697216507, 0.23421564285117952, 0.27490744181415, 0.30152097794928107,
0.3368886210873086, 0.29109625864051736, 0.15918079718341788]
```

[25]:
```
avg_metrics = [y/5 for y in [sum(x) for x in zip(nmf_hs, nmf_cs, nmf_vs,␣
 ↪nmf_ari, nmf_ms)]]
best_r_NMF = r[avg_metrics.index(max(avg_metrics))]
```

```
print('Best value of r for NMF (according to avg. metric): ', best_r_NMF, ',␣
 ↪avg. value of 5 metrics: ',max(avg_metrics))
```

```
Best value of r for NMF (according to avg. metric):  20 , avg. value of 5
metrics:  0.29113664612512097
```

[26]:
```
nmf_normal = NMF(n_components=best_r_NMF,init='random', random_state=1,␣
 ↪max_iter=1000)
nmf_km_normal = nmf_normal.fit_transform(data_feat)
kmean_nmf_normal= km.fit(nmf_km_normal)

nmf_KL = NMF(n_components=best_r_NMF,init='random', random_state=1,
          beta_loss='kullback-leibler', solver='mu', max_iter=1000,alpha=.1,
          l1_ratio=.5)
nmf_km_KL = nmf_KL.fit_transform(data_feat)
kmean_nmf_KL = km.fit(nmf_km_KL)

print("Homogeneity (NMF Normal, best r): %0.3f" % homogeneity_score(dataset.
 ↪target, kmean_nmf_normal.labels_))
print("Completeness (NMF Normal, best r): %0.3f" % completeness_score(dataset.
 ↪target, kmean_nmf_normal.labels_))
print("V-measure (NMF Normal, best r): %0.3f" % v_measure_score(dataset.target,␣
 ↪kmean_nmf_normal.labels_))
print("Adjusted Rand-Index (NMF Normal, best r): %.3f"%␣
 ↪adjusted_rand_score(dataset.target, kmean_nmf_normal.labels_))
print("Adjusted Mutual Information Score (NMF Normal, best r): %.3f"%␣
 ↪adjusted_mutual_info_score(dataset.target, kmean_nmf_normal.labels_))

print("Homogeneity (NMF KL, best r): %0.3f" % homogeneity_score(dataset.target,␣
 ↪kmean_nmf_KL.labels_))
print("Completeness (NMF KL, best r): %0.3f" % completeness_score(dataset.
 ↪target, kmean_nmf_KL.labels_))
print("V-measure (NMF KL, best r): %0.3f" % v_measure_score(dataset.target,␣
 ↪kmean_nmf_KL.labels_))
print("Adjusted Rand-Index (NMF KL, best r): %.3f"% adjusted_rand_score(dataset.
 ↪target, kmean_nmf_KL.labels_))
print("Adjusted Mutual Information Score (NMF KL, best r): %.3f"%␣
 ↪adjusted_mutual_info_score(dataset.target, kmean_nmf_KL.labels_))
```

```
Homogeneity (NMF Normal, best r): 0.424
Completeness (NMF Normal, best r): 0.456
V-measure (NMF Normal, best r): 0.439
Adjusted Rand-Index (NMF Normal, best r): 0.214
Adjusted Mutual Information Score (NMF Normal, best r): 0.437
Homogeneity (NMF KL, best r): 0.424
Completeness (NMF KL, best r): 0.456
V-measure (NMF KL, best r): 0.439
```

```
Adjusted Rand-Index (NMF KL, best r): 0.214
Adjusted Mutual Information Score (NMF KL, best r): 0.437
```

0.9 Question 11

[27]:
```
euc_hs = []
euc_cs = []
euc_vs = []
euc_ari = []
euc_ms = []
cos_hs = []
cos_cs = []
cos_vs = []
cos_ari = []
cos_ms = []

for i in range(len(r)-1):
    print('Testing UMAP (euc) for r = ',r[i])
    Umap_euc = umap.UMAP(n_components=r[i], metric='euclidean').
 ↪fit_transform(data_feat)
    kmean_euc = km.fit(Umap_euc)
    euc_hs.append(homogeneity_score(dataset.target, kmean_euc.labels_))
    euc_cs.append(completeness_score(dataset.target, kmean_euc.labels_))
    euc_vs.append(v_measure_score(dataset.target, kmean_euc.labels_))
    euc_ari.append(adjusted_rand_score(dataset.target, kmean_euc.labels_))
    euc_ms.append(adjusted_mutual_info_score(dataset.target, kmean_euc.labels_))

    print('Testing UMAP (cos) for r = ',r[i])
    Umap_cos = umap.UMAP(n_components=r[i], metric='cosine').
 ↪fit_transform(data_feat)
    kmean_cos = km.fit(Umap_cos)
    cos_hs.append(homogeneity_score(dataset.target, kmean_cos.labels_))
    cos_cs.append(completeness_score(dataset.target, kmean_cos.labels_))
    cos_vs.append(v_measure_score(dataset.target, kmean_cos.labels_))
    cos_ari.append(adjusted_rand_score(dataset.target, kmean_cos.labels_))
    cos_ms.append(adjusted_mutual_info_score(dataset.target, kmean_cos.labels_))
print('Done testing')
```

```
Testing UMAP (euc) for r =  1
Testing UMAP (cos) for r =  1
Testing UMAP (euc) for r =  2
Testing UMAP (cos) for r =  2
Testing UMAP (euc) for r =  3
Testing UMAP (cos) for r =  3
Testing UMAP (euc) for r =  5
Testing UMAP (cos) for r =  5
Testing UMAP (euc) for r =  10
```

```
Testing UMAP (cos) for r =  10
Testing UMAP (euc) for r =  20
Testing UMAP (cos) for r =  20
Testing UMAP (euc) for r =  50
Testing UMAP (cos) for r =  50
Testing UMAP (euc) for r =  100
Testing UMAP (cos) for r =  100
Done testing
```

[28]:
```
print('R: ',r[:-1])
print('Homogeneity (UMAP (euclidean), various r): ',euc_hs)
print('Completeness (UMAP (euclidean), various r): ',euc_cs)
print('V-measure (UMAP (euclidean), various r): ',euc_vs)
print('Adjusted Rand-Index (UMAP (euclidean), various r): ',euc_ari)
print('Adjusted Mutual Information Score (UMAP (euclidean), various r):␣
 ↪',euc_ms)
print('Homogeneity (UMAP (cosine), various r): ',cos_hs)
print('Completeness (UMAP (cosine), various r): ',cos_cs)
print('V-measure (UMAP (cosine), various r): ',cos_vs)
print('Adjusted Rand-Index (UMAP (cosine), various r): ',cos_ari)
print('Adjusted Mutual Information Score (UMAP (cosine), various r): ',cos_ms)
```

```
R:  [1, 2, 3, 5, 10, 20, 50, 100]
Homogeneity (UMAP (euclidean), various r):  [0.012620293092117551,
0.008419977534533463, 0.009794116947980226, 0.013679098125032495,
0.013467682294252471, 0.013824941934853181, 0.01458029362754561,
0.011682413554848874]
Completeness (UMAP (euclidean), various r):  [0.014174116763440709,
0.009273328579959345, 0.010247660667052927, 0.014149394821237487,
0.014484819777446695, 0.014565127043529679, 0.015728848995795085,
0.012324864566508007]
V-measure (UMAP (euclidean), various r):  [0.013352151351033468,
0.008826074427056622, 0.010015756979665238, 0.013910272506903899,
0.013957745203043187, 0.014185385446148885, 0.01513280924056315,
0.011995042848723561]
Adjusted Rand-Index (UMAP (euclidean), various r):  [0.0007458948060444592,
0.001044484970607995, 0.0014289351548669924, 0.0021771411090928425,
0.0026508157960271716, 0.002946492750673443, 0.0029617312569419113,
0.0021540715777906656]
Adjusted Mutual Information Score (UMAP (euclidean), various r):
[0.009942042022715867, 0.005459371146616664, 0.006752199982667159,
0.010644416236496132, 0.010695247851365691, 0.01093182608740133,
0.011832127381706347, 0.00873190262106467]
Homogeneity (UMAP (cosine), various r):  [0.38764015955882575,
0.5414937369554365, 0.5503453726847561, 0.5607289493767216, 0.5287266253822892,
0.539615571071752, 0.5475280275109141, 0.5484730686420062]
Completeness (UMAP (cosine), various r):  [0.4012932401021186,
0.5533980464978849, 0.5656415828454279, 0.5660120297949187, 0.5655365194293139,
```

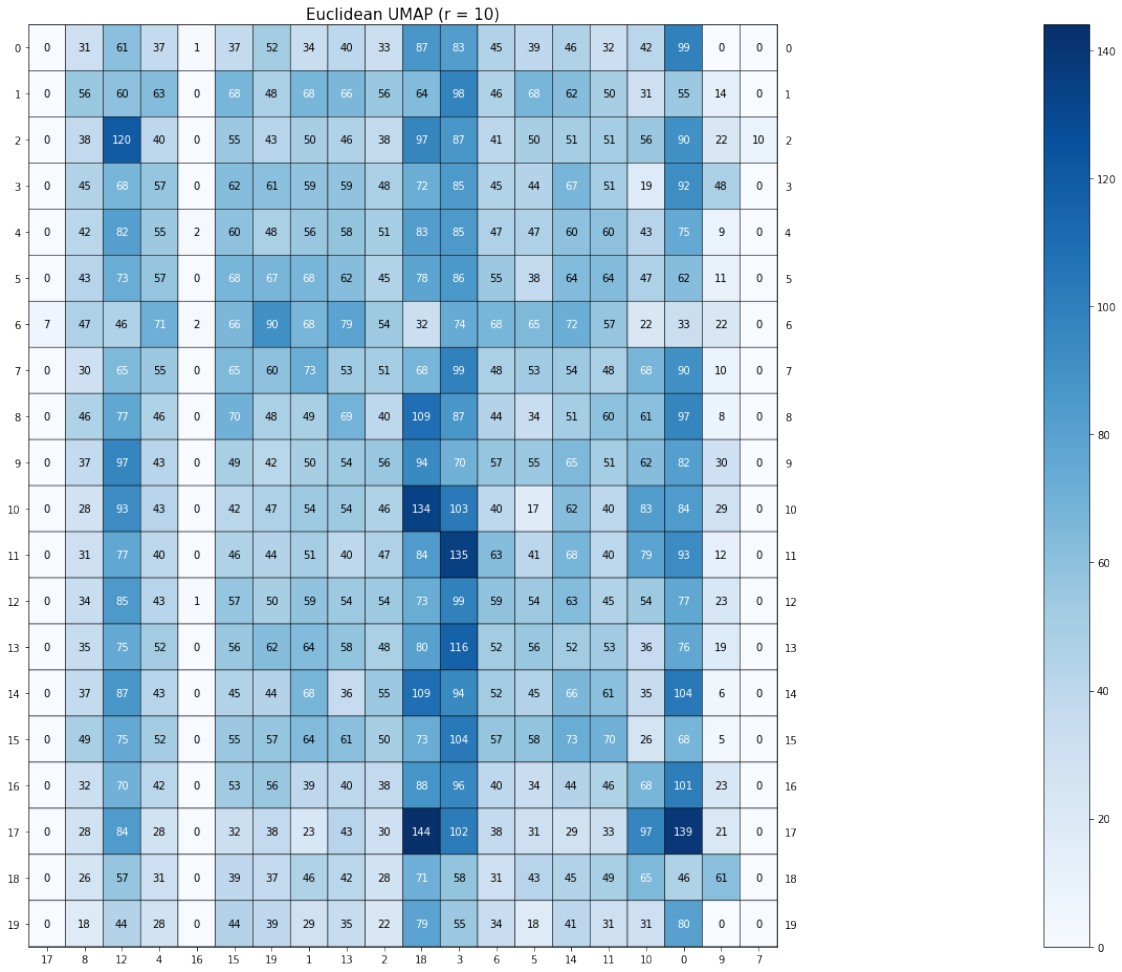
```
0.5711574852601831, 0.58327815865257, 0.5910481376785007]
V-measure (UMAP (cosine), various r):  [0.39434856146264435, 0.5473811763877459,
0.5578886494585783, 0.5633581038914941, 0.5465124487945487, 0.5549386903538244,
0.5648379777276328, 0.5689652531073505]
Adjusted Rand-Index (UMAP (cosine), various r):  [0.25045111085380856,
0.41412727593007953, 0.43361466771646867, 0.44732645053169007,
0.40713388685644797, 0.4137780037369224, 0.42472435424614463,
0.4170597910435684]
Adjusted Mutual Information Score (UMAP (cosine), various r):
[0.39234263463331814, 0.5458918575715307, 0.5564292987847055,
0.5619418336479003, 0.5449837241842901, 0.5534462444823766, 0.563373693165615,
0.5675062694336586]
```

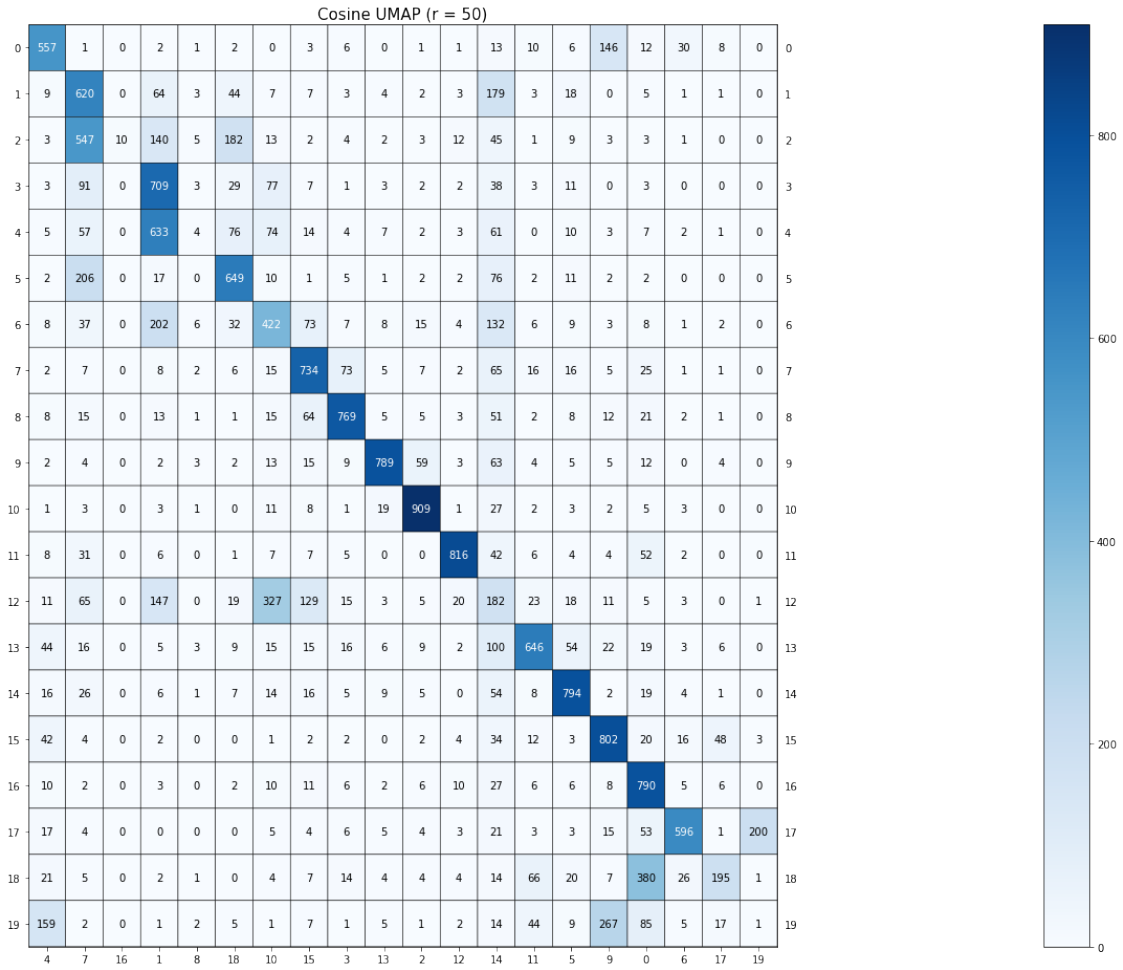
[29]:
```
avg_metrics = [y/5 for y in [sum(x) for x in zip(euc_hs, euc_cs, euc_vs,␣
 ↪euc_ari, euc_ms)]]
best_r_euc = r[avg_metrics.index(max(avg_metrics))]
print('Best value of r for Euclidean UMAP (according to avg. metric): ',␣
 ↪best_r_euc, ', avg. value of 5 metrics: ',max(avg_metrics))
print('Metrics: ')
print('Homogeneity (UMAP (euclidean), best r): ',euc_hs[avg_metrics.
 ↪index(max(avg_metrics))])
print('Completeness (UMAP (euclidean), best r): ',euc_cs[avg_metrics.
 ↪index(max(avg_metrics))])
print('V-measure (UMAP (euclidean), best r): ',euc_vs[avg_metrics.
 ↪index(max(avg_metrics))])
print('Adjusted Rand-Index (UMAP (euclidean), best r): ',euc_ari[avg_metrics.
 ↪index(max(avg_metrics))])
print('Adjusted Mutual Information Score (UMAP (euclidean), best r):␣
 ↪',euc_ms[avg_metrics.index(max(avg_metrics))])
avg_metrics = [y/5 for y in [sum(x) for x in zip(cos_hs, cos_cs, cos_vs,␣
 ↪cos_ari, cos_ms)]]
best_r_cos = r[avg_metrics.index(max(avg_metrics))]
print('Best value of r for Cosine UMAP (according to avg. metric): ',␣
 ↪best_r_cos, ', avg. value of 5 metrics: ',max(avg_metrics))
print('Metrics: ')
print('Homogeneity (UMAP (cosine), best r): ',cos_hs[avg_metrics.
 ↪index(max(avg_metrics))])
print('Completeness (UMAP (cosine), best r): ',cos_cs[avg_metrics.
 ↪index(max(avg_metrics))])
print('V-measure (UMAP (cosine), best r): ',cos_vs[avg_metrics.
 ↪index(max(avg_metrics))])
print('Adjusted Rand-Index (UMAP (cosine), best r): ',cos_ari[avg_metrics.
 ↪index(max(avg_metrics))])
print('Adjusted Mutual Information Score (UMAP (cosine), best r):␣
 ↪',cos_ms[avg_metrics.index(max(avg_metrics))])
```

```
Best value of r for Euclidean UMAP (according to avg. metric):  50 , avg. value
```

```
of 5 metrics:  0.012047162100510421
Metrics:
Homogeneity (UMAP (euclidean), best r):  0.01458029362754561
Completeness (UMAP (euclidean), best r):  0.015728848995795085
V-measure (UMAP (euclidean), best r):  0.01513280924056315
Adjusted Rand-Index (UMAP (euclidean), best r):  0.0029617312569419113
Adjusted Mutual Information Score (UMAP (euclidean), best r):
0.011832127381706347
Best value of r for Cosine UMAP (according to avg. metric):  5 , avg. value of 5
metrics:  0.539873473448545
Metrics:
Homogeneity (UMAP (cosine), best r):  0.5607289493767216
Completeness (UMAP (cosine), best r):  0.5660120297949187
V-measure (UMAP (cosine), best r):  0.5633581038914941
Adjusted Rand-Index (UMAP (cosine), best r):  0.44732645053169007
Adjusted Mutual Information Score (UMAP (cosine), best r):  0.5619418336479003
```

[13]:
```
Umap_euc = umap.UMAP(n_components=best_r_euc, metric='euclidean').
 ↪fit_transform(data_feat)
kmean_euc = km.fit(Umap_euc)
cm = confusion_matrix(dataset.target, kmean_euc.labels_)
rows, cols = linear_sum_assignment(cm, maximize=True)
plot_mat(cm[rows[:, np.newaxis], cols], xticklabels=cols, yticklabels=rows,␣
 ↪title = 'Euclidean UMAP (r = 10)', size=(15,13),pic_fname = 'Q111.png')

Umap_cos = umap.UMAP(n_components=best_r_cos, metric='cosine').
 ↪fit_transform(data_feat)
kmean_cos = km.fit(Umap_cos)
cm = confusion_matrix(dataset.target, kmean_cos.labels_)
rows, cols = linear_sum_assignment(cm, maximize=True)
plot_mat(cm[rows[:, np.newaxis], cols], xticklabels=cols, yticklabels=rows,␣
 ↪title = 'Cosine UMAP (r = 50)', size=(15,13),pic_fname = 'Q112.png')
```


Euclidean UMAP (r = 10)
0 0 31 61 37 1 37 52 34 40 33 87 83 45 39 46 32 42 99 0 0 0
1 0 56 60 63 0 68 48 68 66 56 64 98 46 68 62 50 31 55 14 0 1
2 0 38 120 40 0 55 43 50 46 38 97 87 41 50 51 51 56 90 22 10 2
3 0 45 68 57 0 62 61 59 59 48 72 85 45 44 67 51 19 92 48 0 3
4 0 42 82 55 2 60 48 56 58 51 83 85 47 47 60 60 43 75 9 0 4
5 0 43 73 57 0 68 67 68 62 45 78 86 55 38 64 64 47 62 11 0 5
6 7 47 46 71 2 66 90 68 79 54 32 74 68 65 72 57 22 33 22 0 6
7 0 30 65 55 0 65 60 73 53 51 68 99 48 53 54 48 68 90 10 0 7
8 0 46 77 46 0 70 48 49 69 40 109 87 44 34 51 60 61 97 8 0 8
9 0 37 97 43 0 49 42 50 54 56 94 70 57 55 65 51 62 82 30 0 9
10 0 28 93 43 0 42 47 54 54 46 134 103 40 17 62 40 83 84 29 0 10
11 0 31 77 40 0 46 44 51 40 47 84 135 63 41 68 40 79 93 12 0 11
12 0 34 85 43 1 57 50 59 54 54 73 99 59 54 63 45 54 77 23 0 12
13 0 35 75 52 0 56 62 64 58 48 80 116 52 56 52 53 36 76 19 0 13
14 0 37 87 43 0 45 44 68 36 55 109 94 52 45 66 61 35 104 6 0 14
15 0 49 75 52 0 55 57 64 61 50 73 104 57 58 73 70 26 68 5 0 15
16 0 32 70 42 0 53 56 39 40 38 88 96 40 34 44 46 68 101 23 0 16
17 0 28 84 28 0 32 38 23 43 30 144 102 38 31 29 33 97 139 21 0 17
18 0 26 57 31 0 39 37 46 42 28 71 58 31 43 45 49 65 46 61 0 18
19 0 18 44 28 0 44 39 29 35 22 79 55 34 18 41 31 31 80 0 0 19
17 8 12 4 16 15 19 1 13 2 18 3 6 5 14 11 10 0 9 7
140
120
100
80
60
40
20
0



0.10 Question 13

[31]:
```
ac_w = AgglomerativeClustering(n_clusters=20, linkage='ward').fit(Umap_cos)
ac_s = AgglomerativeClustering(n_clusters=20, linkage='single').fit(Umap_cos)
print("Agglomerative Clustering, Ward - Homogeneity: %0.3f" %␣
 ↪homogeneity_score(dataset.target, ac_w.labels_))
print("Agglomerative Clustering, Ward - Completeness: %0.3f" %␣
 ↪completeness_score(dataset.target, ac_w.labels_))
print("Agglomerative Clustering, Ward - V-measure: %0.3f" %␣
 ↪v_measure_score(dataset.target, ac_w.labels_))
print("Agglomerative Clustering, Ward - Adjusted Rand-Index: %.3f"%␣
 ↪adjusted_rand_score(dataset.target, ac_w.labels_))
print("Agglomerative Clustering, Ward - Adjusted Mutual Information Score: %.
 ↪3f"% adjusted_mutual_info_score(dataset.target, ac_w.labels_))
print()
```

```
print("Agglomerative Clustering, Single - Homogeneity: %0.3f" %␣
 ↪homogeneity_score(dataset.target, ac_s.labels_))
print("Agglomerative Clustering, Single - Completeness: %0.3f" %␣
 ↪completeness_score(dataset.target, ac_s.labels_))
print("Agglomerative Clustering, Single - V-measure: %0.3f" %␣
 ↪v_measure_score(dataset.target, ac_s.labels_))
print("Agglomerative Clustering, Single - Adjusted Rand-Index: %.3f"%␣
 ↪adjusted_rand_score(dataset.target, ac_s.labels_))
print("Agglomerative Clustering, Single - Adjusted Mutual Information Score: %.
 ↪3f"% adjusted_mutual_info_score(dataset.target, ac_s.labels_))
```

```
Agglomerative Clustering, Ward - Homogeneity: 0.539
Agglomerative Clustering, Ward - Completeness: 0.560
Agglomerative Clustering, Ward - V-measure: 0.550
Agglomerative Clustering, Ward - Adjusted Rand-Index: 0.406
Agglomerative Clustering, Ward - Adjusted Mutual Information Score: 0.548

Agglomerative Clustering, Single - Homogeneity: 0.016
Agglomerative Clustering, Single - Completeness: 0.410
Agglomerative Clustering, Single - V-measure: 0.032
Agglomerative Clustering, Single - Adjusted Rand-Index: 0.000
Agglomerative Clustering, Single - Adjusted Mutual Information Score: 0.027
```

0.11 Question 14

[32]:
```
eps_rec = []
min_samples_rec = []
db_hs = []
db_cs = []
db_vs = []
db_ari = []
db_ms = []

eps = [0.01,0.05, 0.1, 0.15, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0,3.0,5.
 ↪0,10.0,30.0,50.0]
min_samples = [5,15,30,60,100,200,500,1000,3000]

for i in range(len(eps)):
    for j in range(len(min_samples)):
        print('Testing for e and min_sample = ',eps[i],'and',min_samples[j])
        dbs = DBSCAN(eps=eps[i],min_samples=min_samples[j],n_jobs=-1).
 ↪fit_predict(Umap_cos)
        db_hs.append(homogeneity_score(dataset.target, dbs))
        db_cs.append(completeness_score(dataset.target, dbs))
        db_vs.append(v_measure_score(dataset.target, dbs))
        db_ari.append(adjusted_rand_score(dataset.target, dbs))
        db_ms.append(adjusted_mutual_info_score(dataset.target, dbs))
```

```
        eps_rec.append(eps[i])
        min_samples_rec.append(min_samples[j])
print('Done testing')
```

```
Testing for e and min_sample =  0.01 and 5
Testing for e and min_sample =  0.01 and 15
Testing for e and min_sample =  0.01 and 30
Testing for e and min_sample =  0.01 and 60
Testing for e and min_sample =  0.01 and 100
Testing for e and min_sample =  0.01 and 200
Testing for e and min_sample =  0.01 and 500
Testing for e and min_sample =  0.01 and 1000
Testing for e and min_sample =  0.01 and 3000
Testing for e and min_sample =  0.05 and 5
Testing for e and min_sample =  0.05 and 15
Testing for e and min_sample =  0.05 and 30
Testing for e and min_sample =  0.05 and 60
Testing for e and min_sample =  0.05 and 100
Testing for e and min_sample =  0.05 and 200
Testing for e and min_sample =  0.05 and 500
Testing for e and min_sample =  0.05 and 1000
Testing for e and min_sample =  0.05 and 3000
Testing for e and min_sample =  0.1 and 5
Testing for e and min_sample =  0.1 and 15
Testing for e and min_sample =  0.1 and 30
Testing for e and min_sample =  0.1 and 60
Testing for e and min_sample =  0.1 and 100
Testing for e and min_sample =  0.1 and 200
Testing for e and min_sample =  0.1 and 500
Testing for e and min_sample =  0.1 and 1000
Testing for e and min_sample =  0.1 and 3000
Testing for e and min_sample =  0.15 and 5
Testing for e and min_sample =  0.15 and 15
Testing for e and min_sample =  0.15 and 30
Testing for e and min_sample =  0.15 and 60
Testing for e and min_sample =  0.15 and 100
Testing for e and min_sample =  0.15 and 200
Testing for e and min_sample =  0.15 and 500
Testing for e and min_sample =  0.15 and 1000
Testing for e and min_sample =  0.15 and 3000
Testing for e and min_sample =  0.2 and 5
Testing for e and min_sample =  0.2 and 15
Testing for e and min_sample =  0.2 and 30
Testing for e and min_sample =  0.2 and 60
Testing for e and min_sample =  0.2 and 100
Testing for e and min_sample =  0.2 and 200
Testing for e and min_sample =  0.2 and 500
```

```
Testing for e and min_sample =  0.2 and 1000
Testing for e and min_sample =  0.2 and 3000
Testing for e and min_sample =  0.3 and 5
Testing for e and min_sample =  0.3 and 15
Testing for e and min_sample =  0.3 and 30
Testing for e and min_sample =  0.3 and 60
Testing for e and min_sample =  0.3 and 100
Testing for e and min_sample =  0.3 and 200
Testing for e and min_sample =  0.3 and 500
Testing for e and min_sample =  0.3 and 1000
Testing for e and min_sample =  0.3 and 3000
Testing for e and min_sample =  0.4 and 5
Testing for e and min_sample =  0.4 and 15
Testing for e and min_sample =  0.4 and 30
Testing for e and min_sample =  0.4 and 60
Testing for e and min_sample =  0.4 and 100
Testing for e and min_sample =  0.4 and 200
Testing for e and min_sample =  0.4 and 500
Testing for e and min_sample =  0.4 and 1000
Testing for e and min_sample =  0.4 and 3000
Testing for e and min_sample =  0.5 and 5
Testing for e and min_sample =  0.5 and 15
Testing for e and min_sample =  0.5 and 30
Testing for e and min_sample =  0.5 and 60
Testing for e and min_sample =  0.5 and 100
Testing for e and min_sample =  0.5 and 200
Testing for e and min_sample =  0.5 and 500
Testing for e and min_sample =  0.5 and 1000
Testing for e and min_sample =  0.5 and 3000
Testing for e and min_sample =  0.6 and 5
Testing for e and min_sample =  0.6 and 15
Testing for e and min_sample =  0.6 and 30
Testing for e and min_sample =  0.6 and 60
Testing for e and min_sample =  0.6 and 100
Testing for e and min_sample =  0.6 and 200
Testing for e and min_sample =  0.6 and 500
Testing for e and min_sample =  0.6 and 1000
Testing for e and min_sample =  0.6 and 3000
Testing for e and min_sample =  0.7 and 5
Testing for e and min_sample =  0.7 and 15
Testing for e and min_sample =  0.7 and 30
Testing for e and min_sample =  0.7 and 60
Testing for e and min_sample =  0.7 and 100
Testing for e and min_sample =  0.7 and 200
Testing for e and min_sample =  0.7 and 500
Testing for e and min_sample =  0.7 and 1000
Testing for e and min_sample =  0.7 and 3000
Testing for e and min_sample =  0.8 and 5
```

```
Testing for e and min_sample =  0.8 and 15
Testing for e and min_sample =  0.8 and 30
Testing for e and min_sample =  0.8 and 60
Testing for e and min_sample =  0.8 and 100
Testing for e and min_sample =  0.8 and 200
Testing for e and min_sample =  0.8 and 500
Testing for e and min_sample =  0.8 and 1000
Testing for e and min_sample =  0.8 and 3000
Testing for e and min_sample =  0.9 and 5
Testing for e and min_sample =  0.9 and 15
Testing for e and min_sample =  0.9 and 30
Testing for e and min_sample =  0.9 and 60
Testing for e and min_sample =  0.9 and 100
Testing for e and min_sample =  0.9 and 200
Testing for e and min_sample =  0.9 and 500
Testing for e and min_sample =  0.9 and 1000
Testing for e and min_sample =  0.9 and 3000
Testing for e and min_sample =  1.0 and 5
Testing for e and min_sample =  1.0 and 15
Testing for e and min_sample =  1.0 and 30
Testing for e and min_sample =  1.0 and 60
Testing for e and min_sample =  1.0 and 100
Testing for e and min_sample =  1.0 and 200
Testing for e and min_sample =  1.0 and 500
Testing for e and min_sample =  1.0 and 1000
Testing for e and min_sample =  1.0 and 3000
Testing for e and min_sample =  3.0 and 5
Testing for e and min_sample =  3.0 and 15
Testing for e and min_sample =  3.0 and 30
Testing for e and min_sample =  3.0 and 60
Testing for e and min_sample =  3.0 and 100
Testing for e and min_sample =  3.0 and 200
Testing for e and min_sample =  3.0 and 500
Testing for e and min_sample =  3.0 and 1000
Testing for e and min_sample =  3.0 and 3000
Testing for e and min_sample =  5.0 and 5
Testing for e and min_sample =  5.0 and 15
Testing for e and min_sample =  5.0 and 30
Testing for e and min_sample =  5.0 and 60
Testing for e and min_sample =  5.0 and 100
Testing for e and min_sample =  5.0 and 200
Testing for e and min_sample =  5.0 and 500
Testing for e and min_sample =  5.0 and 1000
Testing for e and min_sample =  5.0 and 3000
Testing for e and min_sample =  10.0 and 5
Testing for e and min_sample =  10.0 and 15
Testing for e and min_sample =  10.0 and 30
Testing for e and min_sample =  10.0 and 60
```

```
Testing for e and min_sample =  10.0 and 100
Testing for e and min_sample =  10.0 and 200
Testing for e and min_sample =  10.0 and 500
Testing for e and min_sample =  10.0 and 1000
Testing for e and min_sample =  10.0 and 3000
Testing for e and min_sample =  30.0 and 5
Testing for e and min_sample =  30.0 and 15
Testing for e and min_sample =  30.0 and 30
Testing for e and min_sample =  30.0 and 60
Testing for e and min_sample =  30.0 and 100
Testing for e and min_sample =  30.0 and 200
Testing for e and min_sample =  30.0 and 500
Testing for e and min_sample =  30.0 and 1000
Testing for e and min_sample =  30.0 and 3000
Testing for e and min_sample =  50.0 and 5
Testing for e and min_sample =  50.0 and 15
Testing for e and min_sample =  50.0 and 30
Testing for e and min_sample =  50.0 and 60
Testing for e and min_sample =  50.0 and 100
Testing for e and min_sample =  50.0 and 200
Testing for e and min_sample =  50.0 and 500
Testing for e and min_sample =  50.0 and 1000
Testing for e and min_sample =  50.0 and 3000
Done testing
```

[33]:
```
avg_metrics = [y/5 for y in [sum(x) for x in zip(db_hs, db_cs, db_vs, db_ari,␣
 ↪db_ms)]]
best_eps_db = eps_rec[avg_metrics.index(max(avg_metrics))]
best_minSample_db = min_samples_rec[avg_metrics.index(max(avg_metrics))]
print('Best value of epsilon and minimum number of samples hyperparameters for␣
 ↪DBSCAN: ', best_eps_db,'and',best_minSample_db, ' respectively, avg. value␣
 ↪of 5 metrics: ',max(avg_metrics))
print('Metrics: ')
print('Homogeneity (DBSCAN, best hyperparameters): ',db_hs[avg_metrics.
 ↪index(max(avg_metrics))])
print('Completeness (DBSCAN, best hyperparameters): ',db_cs[avg_metrics.
 ↪index(max(avg_metrics))])
print('V-measure (DBSCAN, best hyperparameters): ',db_vs[avg_metrics.
 ↪index(max(avg_metrics))])
print('Adjusted Rand-Index (DBSCAN, best hyperparameters): ',db_ari[avg_metrics.
 ↪index(max(avg_metrics))])
print('Adjusted Mutual Information Score (DBSCAN, best hyperparameters):␣
 ↪',db_ms[avg_metrics.index(max(avg_metrics))])
```

```
Best value of epsilon and minimum number of samples hyperparameters for DBSCAN:
0.5 and 60  respectively, avg. value of 5 metrics:  0.47767964522835094
Metrics:
```

```
Homogeneity (DBSCAN, best hyperparameters):  0.46122031931292645
Completeness (DBSCAN, best hyperparameters):  0.5851789552890995
V-measure (DBSCAN, best hyperparameters):  0.515857438292457
Adjusted Rand-Index (DBSCAN, best hyperparameters):  0.311695108903137
Adjusted Mutual Information Score (DBSCAN, best hyperparameters):
0.5144464043441347
```

[34]:
```
eps_rec = []
min_samples_rec = []
hdb_hs = []
hdb_cs = []
hdb_vs = []
hdb_ari = []
hdb_ms = []

for i in range(len(eps)):
    for j in range(len(min_samples)):
        print('Testing for e and min_sample = ',eps[i],'and',min_samples[j])
        hdbs = hdbscan.
 ↪HDBSCAN(min_cluster_size=100,cluster_selection_epsilon=eps[i],min_samples=min_samples[j],co
 ↪fit_predict(Umap_cos)
        hdb_hs.append(homogeneity_score(dataset.target, hdbs))
        hdb_cs.append(completeness_score(dataset.target, hdbs))
        hdb_vs.append(v_measure_score(dataset.target, hdbs))
        hdb_ari.append(adjusted_rand_score(dataset.target, hdbs))
        hdb_ms.append(adjusted_mutual_info_score(dataset.target, hdbs))
        eps_rec.append(eps[i])
        min_samples_rec.append(min_samples[j])
print('Done testing')
```

```
Testing for e and min_sample =  0.01 and 5
Testing for e and min_sample =  0.01 and 15
Testing for e and min_sample =  0.01 and 30
Testing for e and min_sample =  0.01 and 60
Testing for e and min_sample =  0.01 and 100
Testing for e and min_sample =  0.01 and 200
Testing for e and min_sample =  0.01 and 500
Testing for e and min_sample =  0.01 and 1000
Testing for e and min_sample =  0.01 and 3000
Testing for e and min_sample =  0.05 and 5
Testing for e and min_sample =  0.05 and 15
Testing for e and min_sample =  0.05 and 30
Testing for e and min_sample =  0.05 and 60
Testing for e and min_sample =  0.05 and 100
Testing for e and min_sample =  0.05 and 200
Testing for e and min_sample =  0.05 and 500
Testing for e and min_sample =  0.05 and 1000
Testing for e and min_sample =  0.05 and 3000
```

```
Testing for e and min_sample =  0.1 and 5
Testing for e and min_sample =  0.1 and 15
Testing for e and min_sample =  0.1 and 30
Testing for e and min_sample =  0.1 and 60
Testing for e and min_sample =  0.1 and 100
Testing for e and min_sample =  0.1 and 200
Testing for e and min_sample =  0.1 and 500
Testing for e and min_sample =  0.1 and 1000
Testing for e and min_sample =  0.1 and 3000
Testing for e and min_sample =  0.15 and 5
Testing for e and min_sample =  0.15 and 15
Testing for e and min_sample =  0.15 and 30
Testing for e and min_sample =  0.15 and 60
Testing for e and min_sample =  0.15 and 100
Testing for e and min_sample =  0.15 and 200
Testing for e and min_sample =  0.15 and 500
Testing for e and min_sample =  0.15 and 1000
Testing for e and min_sample =  0.15 and 3000
Testing for e and min_sample =  0.2 and 5
Testing for e and min_sample =  0.2 and 15
Testing for e and min_sample =  0.2 and 30
Testing for e and min_sample =  0.2 and 60
Testing for e and min_sample =  0.2 and 100
Testing for e and min_sample =  0.2 and 200
Testing for e and min_sample =  0.2 and 500
Testing for e and min_sample =  0.2 and 1000
Testing for e and min_sample =  0.2 and 3000
Testing for e and min_sample =  0.3 and 5
Testing for e and min_sample =  0.3 and 15
Testing for e and min_sample =  0.3 and 30
Testing for e and min_sample =  0.3 and 60
Testing for e and min_sample =  0.3 and 100
Testing for e and min_sample =  0.3 and 200
Testing for e and min_sample =  0.3 and 500
Testing for e and min_sample =  0.3 and 1000
Testing for e and min_sample =  0.3 and 3000
Testing for e and min_sample =  0.4 and 5
Testing for e and min_sample =  0.4 and 15
Testing for e and min_sample =  0.4 and 30
Testing for e and min_sample =  0.4 and 60
Testing for e and min_sample =  0.4 and 100
Testing for e and min_sample =  0.4 and 200
Testing for e and min_sample =  0.4 and 500
Testing for e and min_sample =  0.4 and 1000
Testing for e and min_sample =  0.4 and 3000
Testing for e and min_sample =  0.5 and 5
Testing for e and min_sample =  0.5 and 15
Testing for e and min_sample =  0.5 and 30
```

```
Testing for e and min_sample =  0.5 and 60
Testing for e and min_sample =  0.5 and 100
Testing for e and min_sample =  0.5 and 200
Testing for e and min_sample =  0.5 and 500
Testing for e and min_sample =  0.5 and 1000
Testing for e and min_sample =  0.5 and 3000
Testing for e and min_sample =  0.6 and 5
Testing for e and min_sample =  0.6 and 15
Testing for e and min_sample =  0.6 and 30
Testing for e and min_sample =  0.6 and 60
Testing for e and min_sample =  0.6 and 100
Testing for e and min_sample =  0.6 and 200
Testing for e and min_sample =  0.6 and 500
Testing for e and min_sample =  0.6 and 1000
Testing for e and min_sample =  0.6 and 3000
Testing for e and min_sample =  0.7 and 5
Testing for e and min_sample =  0.7 and 15
Testing for e and min_sample =  0.7 and 30
Testing for e and min_sample =  0.7 and 60
Testing for e and min_sample =  0.7 and 100
Testing for e and min_sample =  0.7 and 200
Testing for e and min_sample =  0.7 and 500
Testing for e and min_sample =  0.7 and 1000
Testing for e and min_sample =  0.7 and 3000
Testing for e and min_sample =  0.8 and 5
Testing for e and min_sample =  0.8 and 15
Testing for e and min_sample =  0.8 and 30
Testing for e and min_sample =  0.8 and 60
Testing for e and min_sample =  0.8 and 100
Testing for e and min_sample =  0.8 and 200
Testing for e and min_sample =  0.8 and 500
Testing for e and min_sample =  0.8 and 1000
Testing for e and min_sample =  0.8 and 3000
Testing for e and min_sample =  0.9 and 5
Testing for e and min_sample =  0.9 and 15
Testing for e and min_sample =  0.9 and 30
Testing for e and min_sample =  0.9 and 60
Testing for e and min_sample =  0.9 and 100
Testing for e and min_sample =  0.9 and 200
Testing for e and min_sample =  0.9 and 500
Testing for e and min_sample =  0.9 and 1000
Testing for e and min_sample =  0.9 and 3000
Testing for e and min_sample =  1.0 and 5
Testing for e and min_sample =  1.0 and 15
Testing for e and min_sample =  1.0 and 30
Testing for e and min_sample =  1.0 and 60
Testing for e and min_sample =  1.0 and 100
Testing for e and min_sample =  1.0 and 200
```

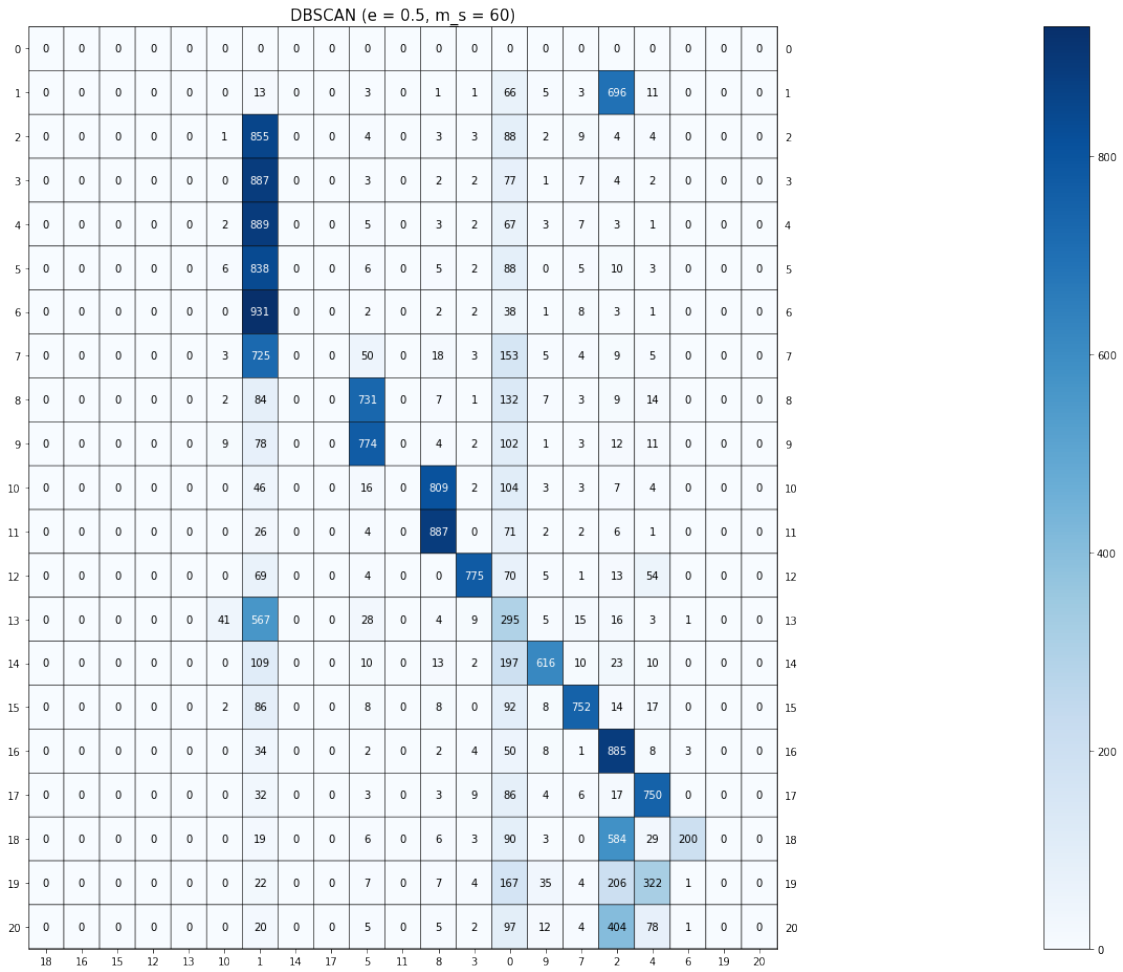
```
Testing for e and min_sample =  1.0 and 500
Testing for e and min_sample =  1.0 and 1000
Testing for e and min_sample =  1.0 and 3000
Testing for e and min_sample =  3.0 and 5
Testing for e and min_sample =  3.0 and 15
Testing for e and min_sample =  3.0 and 30
Testing for e and min_sample =  3.0 and 60
Testing for e and min_sample =  3.0 and 100
Testing for e and min_sample =  3.0 and 200
Testing for e and min_sample =  3.0 and 500
Testing for e and min_sample =  3.0 and 1000
Testing for e and min_sample =  3.0 and 3000
Testing for e and min_sample =  5.0 and 5
Testing for e and min_sample =  5.0 and 15
Testing for e and min_sample =  5.0 and 30
Testing for e and min_sample =  5.0 and 60
Testing for e and min_sample =  5.0 and 100
Testing for e and min_sample =  5.0 and 200
Testing for e and min_sample =  5.0 and 500
Testing for e and min_sample =  5.0 and 1000
Testing for e and min_sample =  5.0 and 3000
Testing for e and min_sample =  10.0 and 5
Testing for e and min_sample =  10.0 and 15
Testing for e and min_sample =  10.0 and 30
Testing for e and min_sample =  10.0 and 60
Testing for e and min_sample =  10.0 and 100
Testing for e and min_sample =  10.0 and 200
Testing for e and min_sample =  10.0 and 500
Testing for e and min_sample =  10.0 and 1000
Testing for e and min_sample =  10.0 and 3000
Testing for e and min_sample =  30.0 and 5
Testing for e and min_sample =  30.0 and 15
Testing for e and min_sample =  30.0 and 30
Testing for e and min_sample =  30.0 and 60
Testing for e and min_sample =  30.0 and 100
Testing for e and min_sample =  30.0 and 200
Testing for e and min_sample =  30.0 and 500
Testing for e and min_sample =  30.0 and 1000
Testing for e and min_sample =  30.0 and 3000
Testing for e and min_sample =  50.0 and 5
Testing for e and min_sample =  50.0 and 15
Testing for e and min_sample =  50.0 and 30
Testing for e and min_sample =  50.0 and 60
Testing for e and min_sample =  50.0 and 100
Testing for e and min_sample =  50.0 and 200
Testing for e and min_sample =  50.0 and 500
Testing for e and min_sample =  50.0 and 1000
Testing for e and min_sample =  50.0 and 3000
```

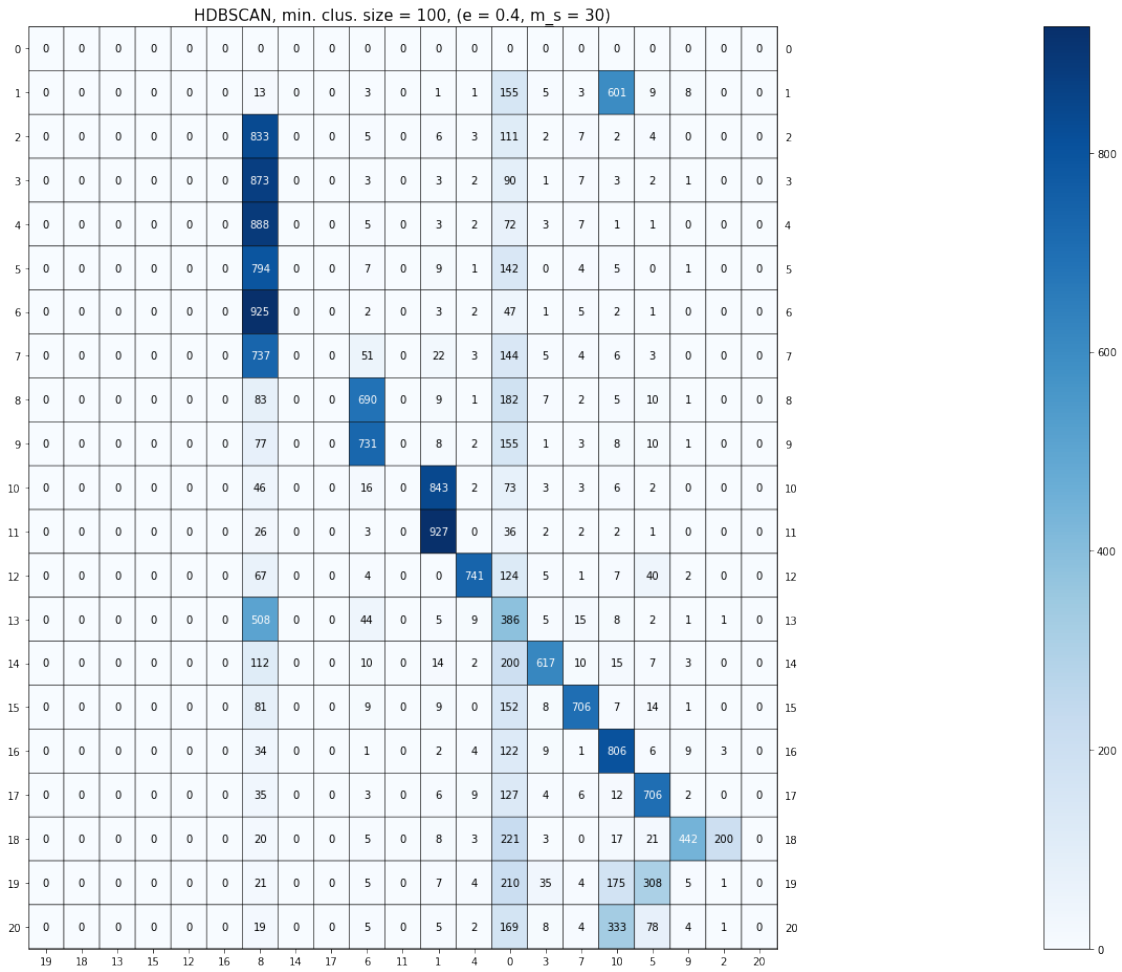
```
Done testing
```

[35]:
```
avg_metrics = [y/5 for y in [sum(x) for x in zip(hdb_hs, hdb_cs, hdb_vs,␣
 ↪hdb_ari, hdb_ms)]]
best_eps_hdb = eps_rec[avg_metrics.index(max(avg_metrics))]
best_minSample_hdb = min_samples_rec[avg_metrics.index(max(avg_metrics))]
print('Best value of epsilon and minimum number of samples hyperparameters for␣
 ↪HDBSCAN: ', best_eps_hdb,'and',best_minSample_hdb, ' respectively, avg.␣
 ↪value of 5 metrics: ',max(avg_metrics))
print('Metrics: ')
print('Homogeneity (HDBSCAN, best hyperparameters): ',hdb_hs[avg_metrics.
 ↪index(max(avg_metrics))])
print('Completeness (HDBSCAN, best hyperparameters): ',hdb_cs[avg_metrics.
 ↪index(max(avg_metrics))])
print('V-measure (HDBSCAN, best hyperparameters): ',hdb_vs[avg_metrics.
 ↪index(max(avg_metrics))])
print('Adjusted Rand-Index (HDBSCAN, best hyperparameters):␣
 ↪',hdb_ari[avg_metrics.index(max(avg_metrics))])
print('Adjusted Mutual Information Score (HDBSCAN, best hyperparameters):␣
 ↪',hdb_ms[avg_metrics.index(max(avg_metrics))])
```

```
Best value of epsilon and minimum number of samples hyperparameters for HDBSCAN:
0.4 and 30  respectively, avg. value of 5 metrics:  0.4595878354145637
Metrics:
Homogeneity (HDBSCAN, best hyperparameters):  0.4419241631830412
Completeness (HDBSCAN, best hyperparameters):  0.5742278618978046
V-measure (HDBSCAN, best hyperparameters):  0.49946299585514187
Adjusted Rand-Index (HDBSCAN, best hyperparameters):  0.2839171766231235
Adjusted Mutual Information Score (HDBSCAN, best hyperparameters):
0.4984069795137075
```

[15]:
```
dbs = DBSCAN(eps=best_eps_db,min_samples=best_minSample_db ,n_jobs=-1).
 ↪fit_predict(Umap_cos)
cm = confusion_matrix(dataset.target, dbs)
rows, cols = linear_sum_assignment(cm, maximize=True)
plot_mat(cm[rows[:, np.newaxis], cols], xticklabels=cols, yticklabels=rows,␣
 ↪title = 'DBSCAN (e = 0.5, m_s = 60)', size=(15,13),pic_fname = 'Q151.png')

hdbs = hdbscan.
 ↪HDBSCAN(min_cluster_size=100,cluster_selection_epsilon=best_eps_hdb,min_samples=best_minSam
 ↪fit_predict(Umap_cos)
cm = confusion_matrix(dataset.target, hdbs)
rows, cols = linear_sum_assignment(cm, maximize=True)
plot_mat(cm[rows[:, np.newaxis], cols], xticklabels=cols, yticklabels=rows,␣
 ↪title = 'HDBSCAN, min. clus. size = 100, (e = 0.4, m_s = 30)',␣
 ↪size=(15,13),pic_fname = 'Q152.png')
```


DBSCAN (e = 0.5, m_s = 60)
18 16 15 12 13 10 1 14 17 5 11 8 3 0 9 7 2 4 6 19 20
0
200
400
600
800



0.12 Question 16

Step 1: Dataset Acquiring and Preprocessing

[2]:
```
#dataset downloaded and stored in the project directory inside "learn-ai-bbc" folder
BBC_dataset_train = pd.read_csv("learn-ai-bbc/BBC News Train.csv",usecols=['Text','Category'])
BBC_dataset_labels = BBC_dataset_train.pop('Category').values
X_BBC=["".join(text) for text in BBC_dataset_train['Text'].values]
y_GT = pd.factorize(BBC_dataset_labels)[0]
print('Size of training set: ',BBC_dataset_train.shape)
print('Size of training set (list): ',len(X_BBC))
print('Size of labels: ',BBC_dataset_labels.shape)
print('Size of labels (factorized to int): ',y_GT.shape)
print('Name of all unique lables ',set(BBC_dataset_labels))
print('Name of all unique lables (factorized to int) ',set(y_GT))
```

```
Size of training set:  (1490, 1)
Size of training set (list):  1490
Size of labels:  (1490,)
Size of labels (factorized to int):  (1490,)
Name of all unique lables  {'politics', 'sport', 'business', 'entertainment',
'tech'}
Name of all unique lables (factorized to int)  {0, 1, 2, 3, 4}
```

Step 2: Feature Extraction

[20]:
```
tfidf_transformer = TfidfVectorizer(min_df=3, stop_words='english')
data_feat = tfidf_transformer.fit_transform(X_BBC)
print(data_feat.shape)
```

```
(1490, 10281)
```

Step 3 and 4: Testing out various dimensionality reduction schemes and clustering

a. K-means with SVD, NMF, UMAP (Euclidean) and UMAP (Cosine)

[39]:
```
svd_hs = []
svd_cs = []
svd_vs = []
svd_ari = []
svd_ms = []
nmf_hs = []
nmf_cs = []
nmf_vs = []
nmf_ari = []
nmf_ms = []
cos_hs = []
cos_cs = []
cos_vs = []
cos_ari = []
cos_ms = []
euc_hs = []
euc_cs = []
euc_vs = []
euc_ari = []
euc_ms = []

r = [1,2,3,5,10,20,50,100]
km = KMeans(n_clusters=5, init='k-means++', max_iter=1000, n_init=30,␣
 ↪random_state=0,n_jobs=-1)

for i in range(len(r)):
    print('Testing SVD for r = ',r[i])
    svd = TruncatedSVD(n_components=r[i], random_state=0)
```

```
    svd_km = svd.fit_transform(data_feat)
    kmean_svd = km.fit(svd_km)
    svd_hs.append(homogeneity_score(y_GT, kmean_svd.labels_))
    svd_cs.append(completeness_score(y_GT, kmean_svd.labels_))
    svd_vs.append(v_measure_score(y_GT, kmean_svd.labels_))
    svd_ari.append(adjusted_rand_score(y_GT, kmean_svd.labels_))
    svd_ms.append(adjusted_mutual_info_score(y_GT, kmean_svd.labels_))
    print('Testing NMF for r = ',r[i])
    nmf = NMF(n_components=r[i], init='random', random_state=0, max_iter=400)
    nmf_km = nmf.fit_transform(data_feat)
    kmean_nmf = km.fit(nmf_km)
    nmf_hs.append(homogeneity_score(y_GT, kmean_nmf.labels_))
    nmf_cs.append(completeness_score(y_GT, kmean_nmf.labels_))
    nmf_vs.append(v_measure_score(y_GT, kmean_nmf.labels_))
    nmf_ari.append(adjusted_rand_score(y_GT, kmean_nmf.labels_))
    nmf_ms.append(adjusted_mutual_info_score(y_GT, kmean_nmf.labels_))
    print('Testing UMAP (cos) for r = ',r[i])
    Umap_cos = umap.UMAP(n_components=r[i], metric='cosine').
↪fit_transform(data_feat)
    kmean_cos = km.fit(Umap_cos)
    cos_hs.append(homogeneity_score(y_GT, kmean_cos.labels_))
    cos_cs.append(completeness_score(y_GT, kmean_cos.labels_))
    cos_vs.append(v_measure_score(y_GT, kmean_cos.labels_))
    cos_ari.append(adjusted_rand_score(y_GT, kmean_cos.labels_))
    cos_ms.append(adjusted_mutual_info_score(y_GT, kmean_cos.labels_))
    print('Testing UMAP (euc) for r = ',r[i])
    Umap_euc = umap.UMAP(n_components=r[i], metric='euclidean').
↪fit_transform(data_feat)
    kmean_euc = km.fit(Umap_euc)
    euc_hs.append(homogeneity_score(y_GT, kmean_euc.labels_))
    euc_cs.append(completeness_score(y_GT, kmean_euc.labels_))
    euc_vs.append(v_measure_score(y_GT, kmean_euc.labels_))
    euc_ari.append(adjusted_rand_score(y_GT, kmean_euc.labels_))
    euc_ms.append(adjusted_mutual_info_score(y_GT, kmean_euc.labels_))
print('Done testing.')
```

```
Testing SVD for r =  1
Testing NMF for r =  1
Testing UMAP (cos) for r =  1
Testing UMAP (euc) for r =  1
Testing SVD for r =  2
Testing NMF for r =  2
Testing UMAP (cos) for r =  2
Testing UMAP (euc) for r =  2
Testing SVD for r =  3
Testing NMF for r =  3
Testing UMAP (cos) for r =  3
```

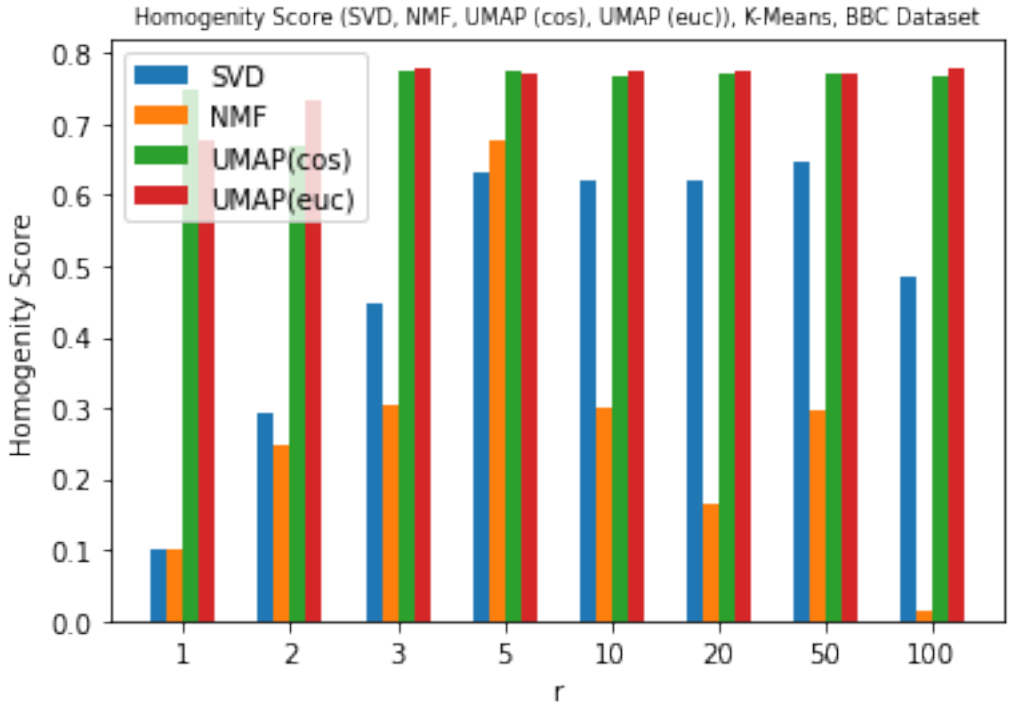
```
Testing UMAP (euc) for r =  3
Testing SVD for r =  5
Testing NMF for r =  5
Testing UMAP (cos) for r =  5
Testing UMAP (euc) for r =  5
Testing SVD for r =  10
Testing NMF for r =  10
Testing UMAP (cos) for r =  10
Testing UMAP (euc) for r =  10
Testing SVD for r =  20
Testing NMF for r =  20
Testing UMAP (cos) for r =  20
Testing UMAP (euc) for r =  20
Testing SVD for r =  50
Testing NMF for r =  50
Testing UMAP (cos) for r =  50
Testing UMAP (euc) for r =  50
Testing SVD for r =  100
Testing NMF for r =  100
Testing UMAP (cos) for r =  100
Testing UMAP (euc) for r =  100
Done testing.
```

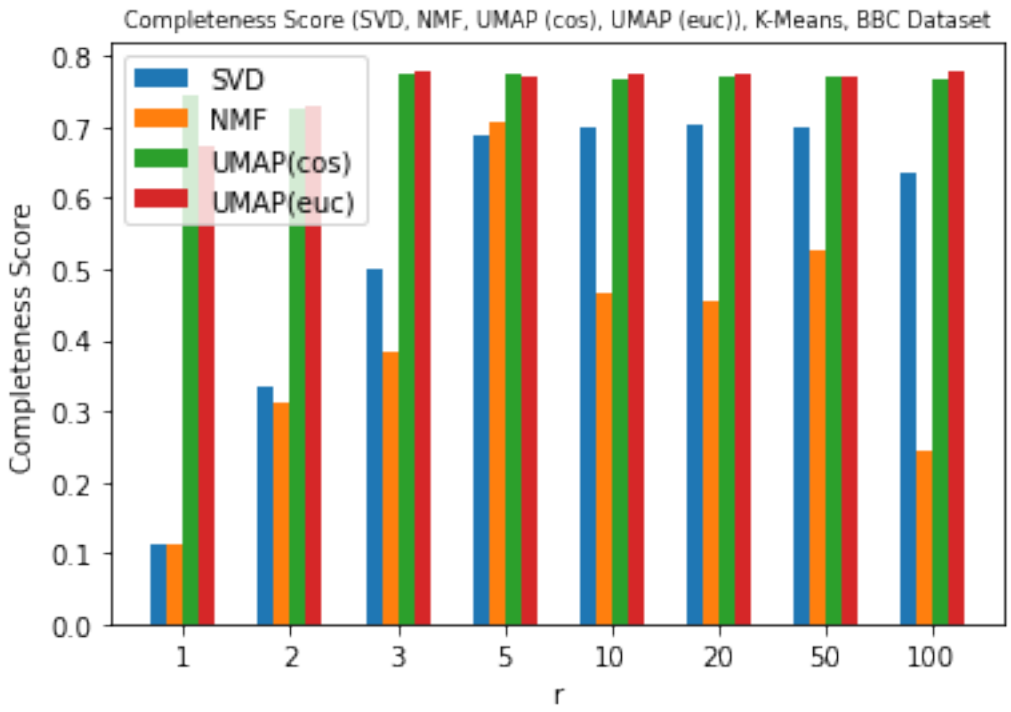
b. Performance Evaluation of K-Means

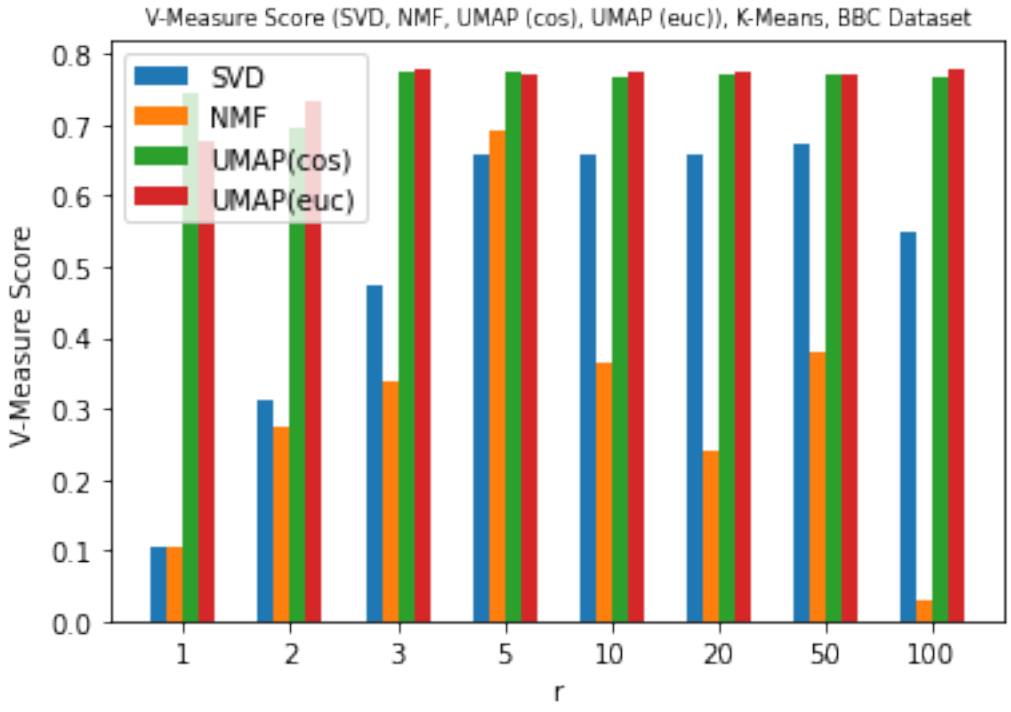
[40]:
```
width = 0.15
fig, ax = plt.subplots()
rects1 = ax.bar(np.arange(len(r)) - 1.5*width, svd_hs, width, label='SVD')
rects2 = ax.bar(np.arange(len(r)) - width/2, nmf_hs, width, label='NMF')
rects3 = ax.bar(np.arange(len(r)) + width/2, cos_hs, width, label='UMAP(cos)')
rects4 = ax.bar(np.arange(len(r)) + 1.5*width, euc_hs, width, label='UMAP(euc)')
ax.set_ylabel('Homogenity Score')
ax.set_title('Homogenity Score (SVD, NMF, UMAP (cos), UMAP (euc)), K-Means, BBC␣
 ↪Dataset',fontsize=8)
ax.set_xticks(np.arange(len(r)))
ax.set_xticklabels(r)
ax.set_xlabel('r')
ax.legend()
plt.savefig('Q161.png',dpi=300,bbox_inches='tight')
plt.show()

width = 0.15
fig, ax = plt.subplots()
rects1 = ax.bar(np.arange(len(r)) - 1.5*width, svd_cs, width, label='SVD')
rects2 = ax.bar(np.arange(len(r)) - width/2, nmf_cs, width, label='NMF')
rects3 = ax.bar(np.arange(len(r)) + width/2, cos_cs, width, label='UMAP(cos)')
rects4 = ax.bar(np.arange(len(r)) + 1.5*width, euc_cs, width, label='UMAP(euc)')
ax.set_ylabel('Completeness Score')
```

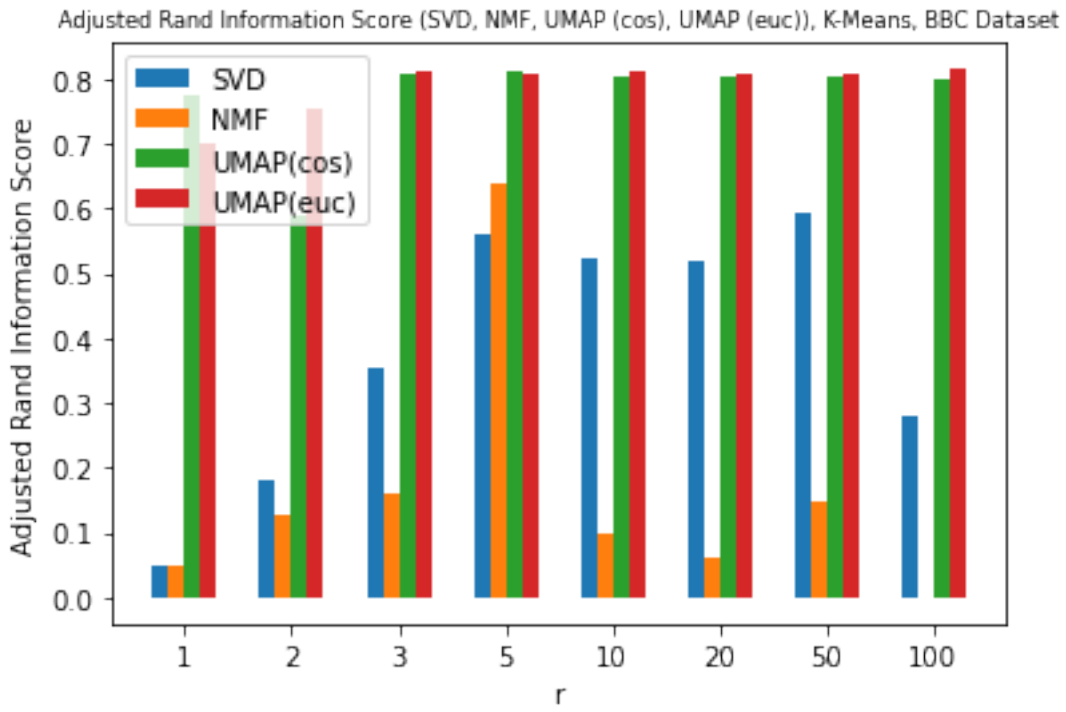
```
ax.set_title('Completeness Score (SVD, NMF, UMAP (cos), UMAP (euc)), K-Means,␣
 ↪BBC Dataset',fontsize=8)
ax.set_xticks(np.arange(len(r)))
ax.set_xticklabels(r)
ax.set_xlabel('r')
ax.legend()
plt.savefig('Q162.png',dpi=300,bbox_inches='tight')
plt.show()

width = 0.15
fig, ax = plt.subplots()
rects1 = ax.bar(np.arange(len(r)) - 1.5*width, svd_vs, width, label='SVD')
rects2 = ax.bar(np.arange(len(r)) - width/2, nmf_vs, width, label='NMF')
rects3 = ax.bar(np.arange(len(r)) + width/2, cos_vs, width, label='UMAP(cos)')
rects4 = ax.bar(np.arange(len(r)) + 1.5*width, euc_vs, width, label='UMAP(euc)')
ax.set_ylabel('V-Measure Score')
ax.set_title('V-Measure Score (SVD, NMF, UMAP (cos), UMAP (euc)), K-Means, BBC␣
 ↪Dataset',fontsize=8)
ax.set_xticks(np.arange(len(r)))
ax.set_xticklabels(r)
ax.set_xlabel('r')
ax.legend()
plt.savefig('Q163.png',dpi=300,bbox_inches='tight')
plt.show()

width = 0.15
fig, ax = plt.subplots()
rects1 = ax.bar(np.arange(len(r)) - 1.5*width, svd_ari, width, label='SVD')
rects2 = ax.bar(np.arange(len(r)) - width/2, nmf_ari, width, label='NMF')
rects3 = ax.bar(np.arange(len(r)) + width/2, cos_ari, width, label='UMAP(cos)')
rects4 = ax.bar(np.arange(len(r)) + 1.5*width, euc_ari, width,␣
 ↪label='UMAP(euc)')
ax.set_ylabel('Adjusted Rand Information Score')
ax.set_title('Adjusted Rand Information Score (SVD, NMF, UMAP (cos), UMAP␣
 ↪(euc)), K-Means, BBC Dataset',fontsize=8)
ax.set_xticks(np.arange(len(r)))
ax.set_xticklabels(r)
ax.set_xlabel('r')
ax.legend()
plt.savefig('Q164.png',dpi=300,bbox_inches='tight')
plt.show()

width = 0.15
fig, ax = plt.subplots()
rects1 = ax.bar(np.arange(len(r)) - 1.5*width, svd_ms, width, label='SVD')
rects2 = ax.bar(np.arange(len(r)) - width/2, nmf_ms, width, label='NMF')
rects3 = ax.bar(np.arange(len(r)) + width/2, cos_ms, width, label='UMAP(cos)')
```

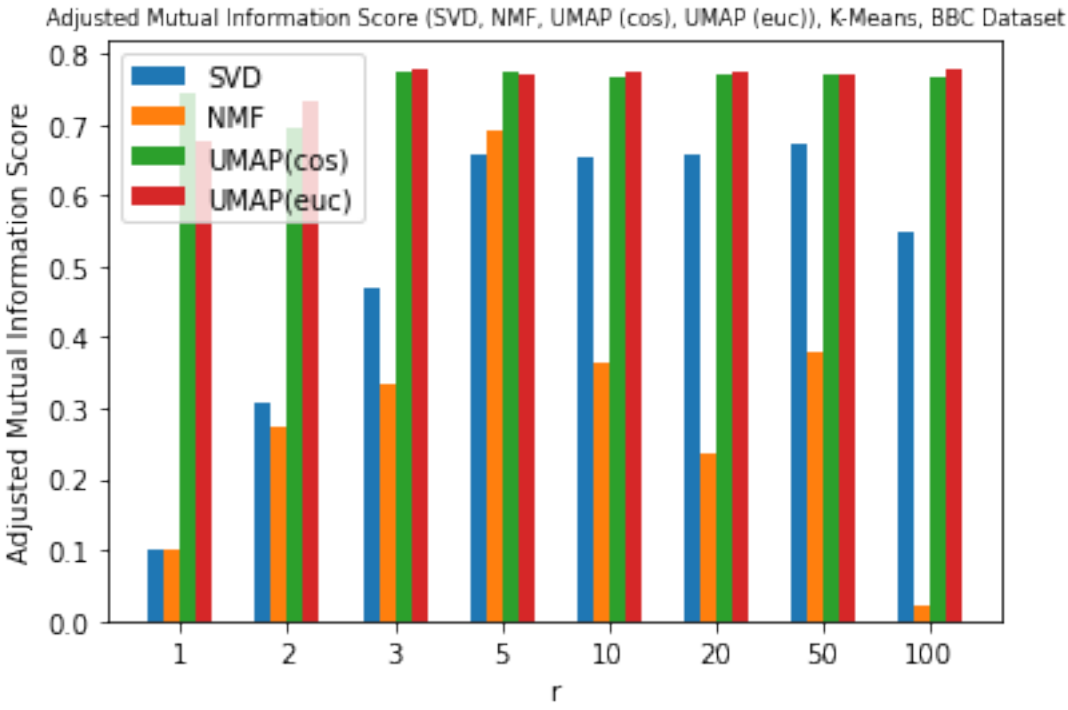
```
rects4 = ax.bar(np.arange(len(r)) + 1.5*width, euc_ms, width, label='UMAP(euc)')
ax.set_ylabel('Adjusted Mutual Information Score')
ax.set_title('Adjusted Mutual Information Score (SVD, NMF, UMAP (cos), UMAP␣
↪(euc)), K-Means, BBC Dataset',fontsize=8)
ax.set_xticks(np.arange(len(r)))
ax.set_xticklabels(r)
ax.set_xlabel('r')
ax.legend()
plt.savefig('Q165.png',dpi=300,bbox_inches='tight')
plt.show()
```



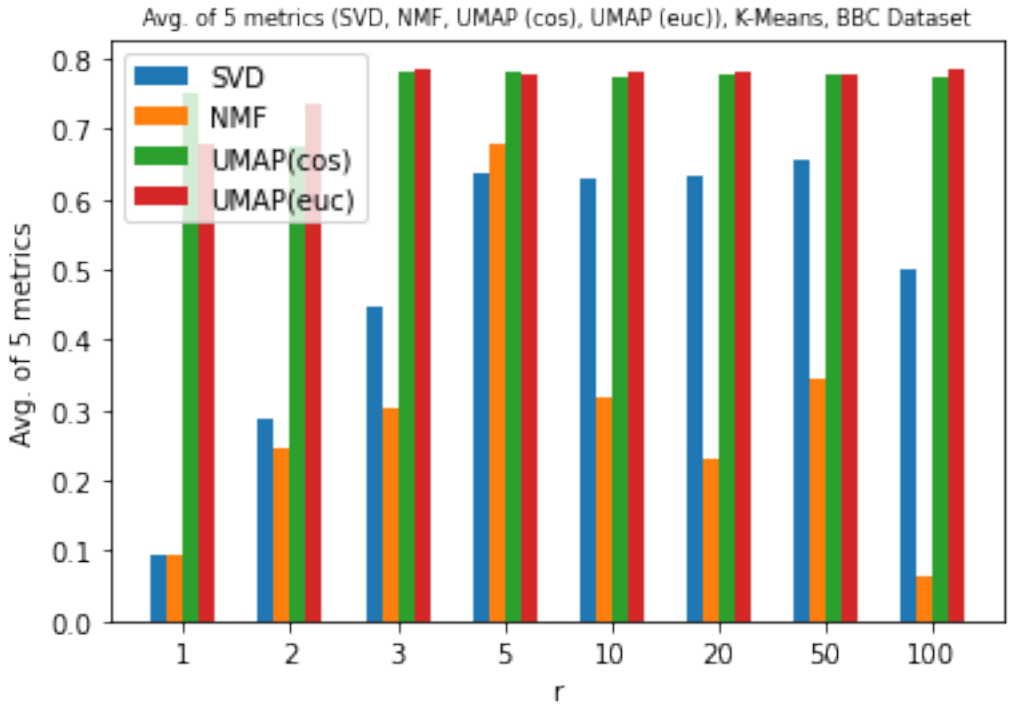

Completeness Score (SVD, NMF, UMAP (cos), UMAP (euc)), K-Means, BBC Dataset
SVD
NMF
UMAP(cos)
UMAP(euc)
Completeness Score
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
1
2
3
5
10
20
50
100
r


V-Measure Score (SVD, NMF, UMAP (cos), UMAP (euc)), K-Means, BBC Dataset
SVD
NMF
UMAP(cos)
UMAP(euc)
V-Measure Score
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
1
2
3
5
10
20
50
100
r


Adjusted Rand Information Score (SVD, NMF, UMAP (cos), UMAP (euc)), K-Means, BBC Dataset
SVD
NMF
UMAP(cos)
UMAP(euc)
Adjusted Rand Information Score
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
1
2
3
5
10
20
50
100
r


Adjusted Mutual Information Score (SVD, NMF, UMAP (cos), UMAP (euc)), K-Means, BBC Dataset
SVD
NMF
UMAP(cos)
UMAP(euc)
Adjusted Mutual Information Score
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
1
2
3
5
10
20
50
100
r

[41]:
```
width = 0.15
fig, ax = plt.subplots()
rects1 = ax.bar(np.arange(len(r)) - 1.5*width, [y/5 for y in [sum(x) for x in␣
 ↪zip(svd_hs, svd_cs, svd_vs, svd_ari, svd_ms)]], width, label='SVD')
rects2 = ax.bar(np.arange(len(r)) - width/2, [y/5 for y in [sum(x) for x in␣
 ↪zip(nmf_hs, nmf_cs, nmf_vs, nmf_ari, nmf_ms)]], width, label='NMF')
rects3 = ax.bar(np.arange(len(r)) + width/2, [y/5 for y in [sum(x) for x in␣
 ↪zip(cos_hs, cos_cs, cos_vs, cos_ari, cos_ms)]], width, label='UMAP(cos)')
rects4 = ax.bar(np.arange(len(r)) + 1.5*width, [y/5 for y in [sum(x) for x in␣
 ↪zip(euc_hs, euc_cs, euc_vs, euc_ari, euc_ms)]], width, label='UMAP(euc)')
ax.set_ylabel('Avg. of 5 metrics')
ax.set_title('Avg. of 5 metrics (SVD, NMF, UMAP (cos), UMAP (euc)), K-Means,␣
 ↪BBC Dataset',fontsize=8)
ax.set_xticks(np.arange(len(r)))
ax.set_xticklabels(r)
ax.set_xlabel('r')
ax.legend()
plt.savefig('Q166.png',dpi=300,bbox_inches='tight')
plt.show()
```



[42]:
```
avg_metrics = [y/5 for y in [sum(x) for x in zip(euc_hs, euc_cs, euc_vs,␣
 ↪euc_ari, euc_ms)]]
best_r_euc = r[avg_metrics.index(max(avg_metrics))]
```

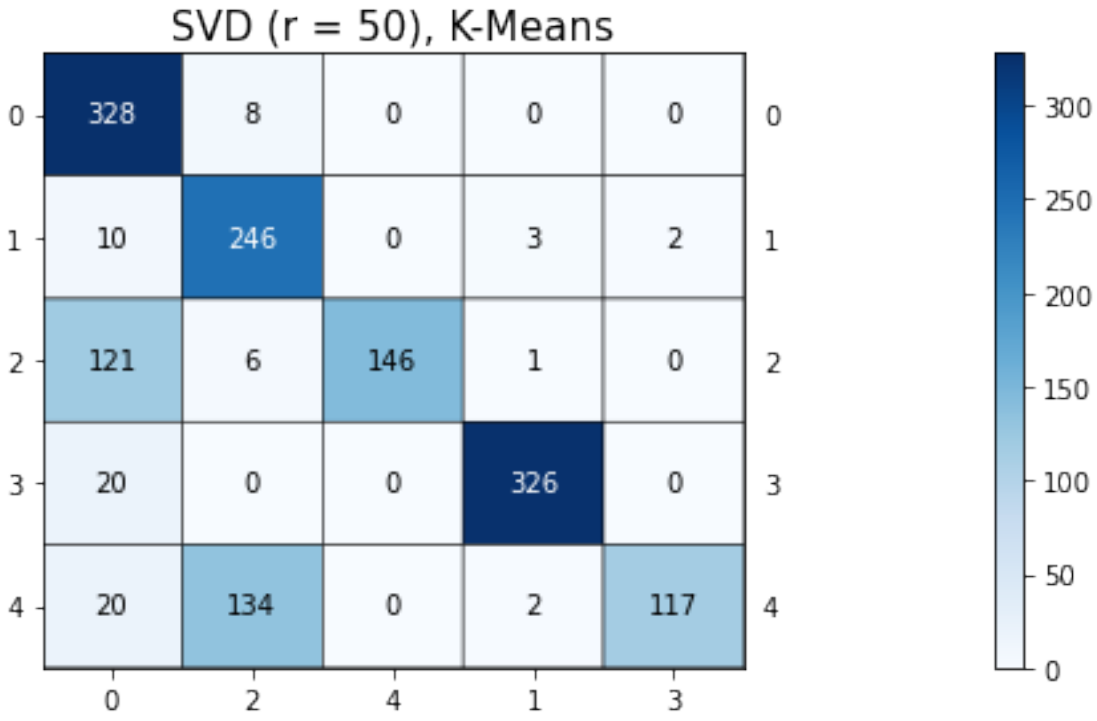
```
print('Best value of r for Euclidean UMAP (according to avg. metric): ',␣
 ↪best_r_euc, ', avg. value of 5 metrics: ',max(avg_metrics))
print('Metrics: ')
print('Homogeneity (UMAP (euclidean), best r): ',euc_hs[avg_metrics.
 ↪index(max(avg_metrics))])
print('Completeness (UMAP (euclidean), best r): ',euc_cs[avg_metrics.
 ↪index(max(avg_metrics))])
print('V-measure (UMAP (euclidean), best r): ',euc_vs[avg_metrics.
 ↪index(max(avg_metrics))])
print('Adjusted Rand-Index (UMAP (euclidean), best r): ',euc_ari[avg_metrics.
 ↪index(max(avg_metrics))])
print('Adjusted Mutual Information Score (UMAP (euclidean), best r):␣
 ↪',euc_ms[avg_metrics.index(max(avg_metrics))])
print('-------------------------------------------------------------------------------
avg_metrics = [y/5 for y in [sum(x) for x in zip(cos_hs, cos_cs, cos_vs,␣
 ↪cos_ari, cos_ms)]]
best_r_cos = r[avg_metrics.index(max(avg_metrics))]
print('Best value of r for Cosine UMAP (according to avg. metric): ',␣
 ↪best_r_cos, ', avg. value of 5 metrics: ',max(avg_metrics))
print('Metrics: ')
print('Homogeneity (UMAP (cosine), best r): ',cos_hs[avg_metrics.
 ↪index(max(avg_metrics))])
print('Completeness (UMAP (cosine), best r): ',cos_cs[avg_metrics.
 ↪index(max(avg_metrics))])
print('V-measure (UMAP (cosine), best r): ',cos_vs[avg_metrics.
 ↪index(max(avg_metrics))])
print('Adjusted Rand-Index (UMAP (cosine), best r): ',cos_ari[avg_metrics.
 ↪index(max(avg_metrics))])
print('Adjusted Mutual Information Score (UMAP (cosine), best r):␣
 ↪',cos_ms[avg_metrics.index(max(avg_metrics))])
print('-------------------------------------------------------------------------------
avg_metrics = [y/5 for y in [sum(x) for x in zip(nmf_hs, nmf_cs, nmf_vs,␣
 ↪nmf_ari, nmf_ms)]]
best_r_NMF = r[avg_metrics.index(max(avg_metrics))]
print('Best value of r for NMF (according to avg. metric): ', best_r_NMF, ',␣
 ↪avg. value of 5 metrics: ',max(avg_metrics))
print('Metrics: ')
print('Homogeneity (NMF, best r): ',nmf_hs[avg_metrics.index(max(avg_metrics))])
print('Completeness (NMF, best r): ',nmf_cs[avg_metrics.
 ↪index(max(avg_metrics))])
print('V-measure (NMF, best r): ',nmf_vs[avg_metrics.index(max(avg_metrics))])
print('Adjusted Rand-Index (NMF, best r): ',nmf_ari[avg_metrics.
 ↪index(max(avg_metrics))])
print('Adjusted Mutual Information Score (NMF, best r): ',nmf_ms[avg_metrics.
 ↪index(max(avg_metrics))])
print('-------------------------------------------------------------------------------
```

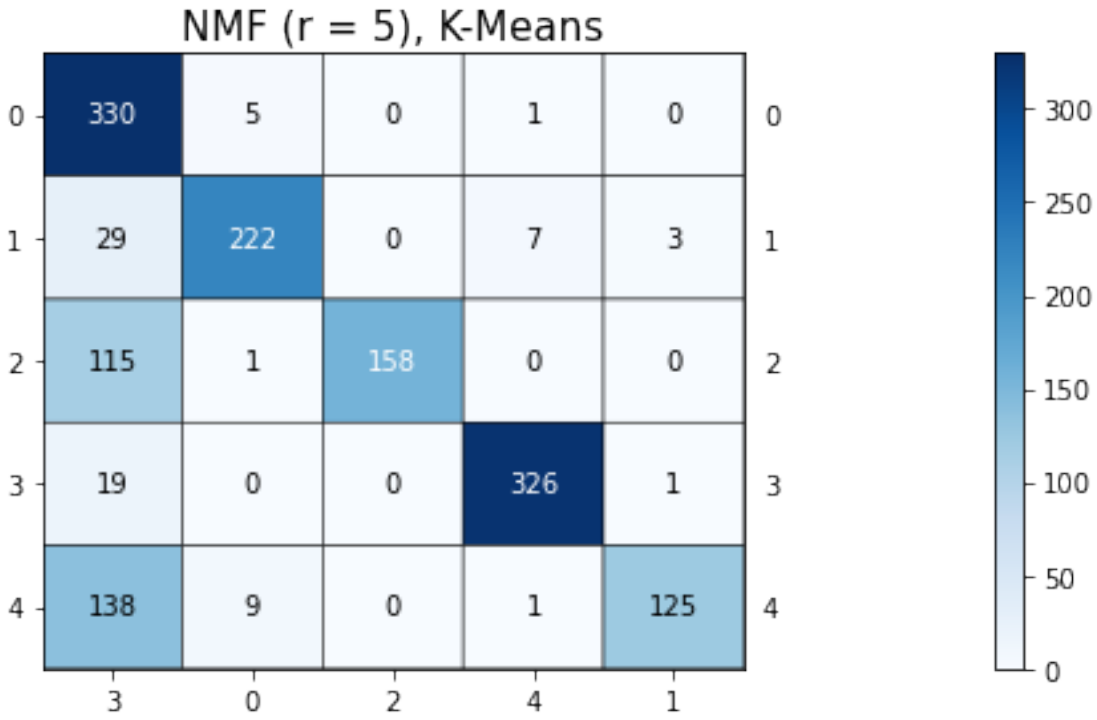
```
avg_metrics = [y/5 for y in [sum(x) for x in zip(svd_hs, svd_cs, svd_vs,␣
 ↪svd_ari, svd_ms)]]
best_r_svd = r[avg_metrics.index(max(avg_metrics))]
print('Best value of r for SVD (according to avg. metric): ', best_r_svd, ',␣
 ↪avg. value of 5 metrics: ',max(avg_metrics))
print('Metrics: ')
print('Homogeneity (SVD, best r): ',svd_hs[avg_metrics.index(max(avg_metrics))])
print('Completeness (SVD, best r): ',svd_cs[avg_metrics.
 ↪index(max(avg_metrics))])
print('V-measure (SVD, best r): ',svd_vs[avg_metrics.index(max(avg_metrics))])
print('Adjusted Rand-Index (SVD, best r): ',svd_ari[avg_metrics.
 ↪index(max(avg_metrics))])
print('Adjusted Mutual Information Score (SVD, best r): ',svd_ms[avg_metrics.
 ↪index(max(avg_metrics))])
```

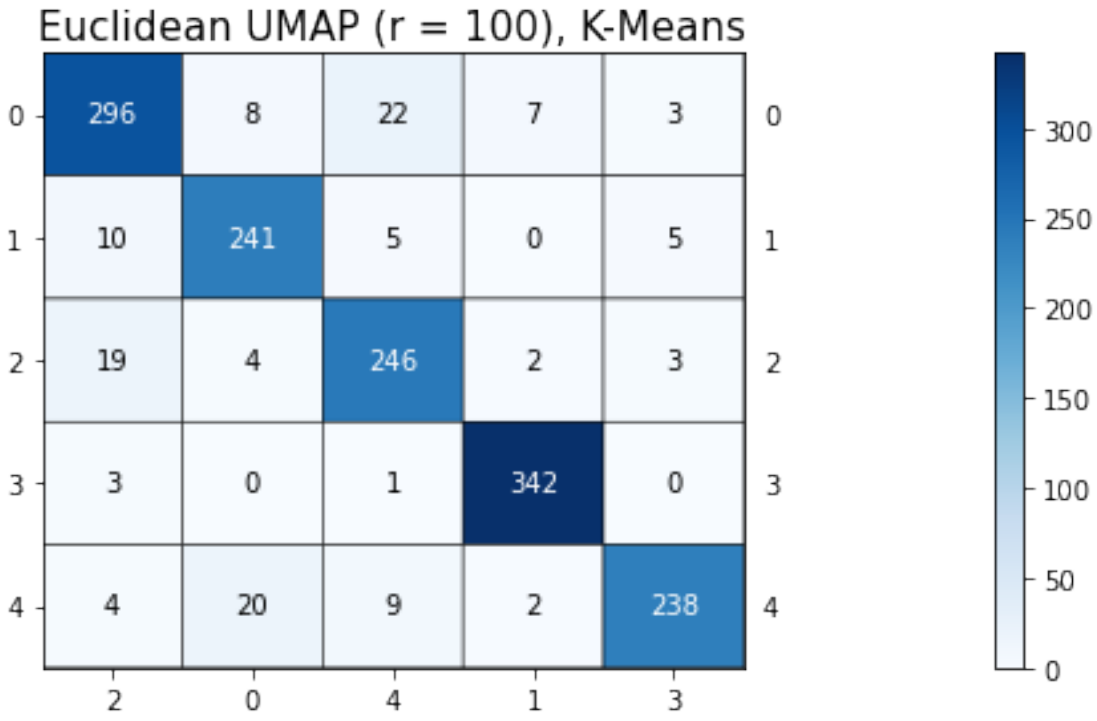
```
Best value of r for Euclidean UMAP (according to avg. metric):  100 , avg. value
of 5 metrics:  0.7863568934147168
Metrics:
Homogeneity (UMAP (euclidean), best r):  0.7792623218537925
Completeness (UMAP (euclidean), best r):  0.7797694659206051
V-measure (UMAP (euclidean), best r):  0.7795158114016759
Adjusted Rand-Index (UMAP (euclidean), best r):  0.8144656785590599
Adjusted Mutual Information Score (UMAP (euclidean), best r):
0.7787711893384506
--------------------------------------------------------------------------------
------------------------
Best value of r for Cosine UMAP (according to avg. metric):  5 , avg. value of 5
metrics:  0.782891909150627
Metrics:
Homogeneity (UMAP (cosine), best r):  0.7759972871042146
Completeness (UMAP (cosine), best r):  0.7762459012114491
V-measure (UMAP (cosine), best r):  0.7761215742482664
Adjusted Rand-Index (UMAP (cosine), best r):  0.8107291656585696
Adjusted Mutual Information Score (UMAP (cosine), best r):  0.7753656175306356
--------------------------------------------------------------------------------
------------------------
Best value of r for NMF (according to avg. metric):  5 , avg. value of 5
metrics:  0.680450511677581
Metrics:
Homogeneity (NMF, best r):  0.6749528460900887
Completeness (NMF, best r):  0.7070646037111105
V-measure (NMF, best r):  0.6906356597928979
Adjusted Rand-Index (NMF, best r):  0.6400335068305302
Adjusted Mutual Information Score (NMF, best r):  0.6895659419632774
--------------------------------------------------------------------------------
------------------------
Best value of r for SVD (according to avg. metric):  50 , avg. value of 5
```

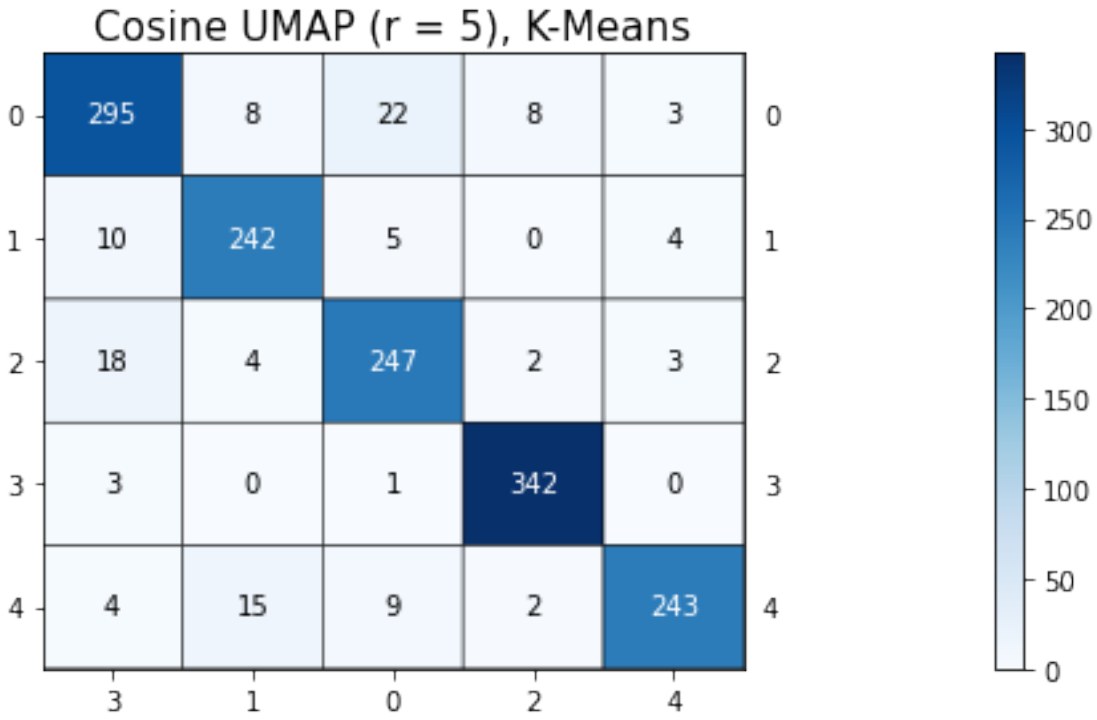
```
metrics:  0.6579695582532739
Metrics:
Homogeneity (SVD, best r):  0.6482818338095657
Completeness (SVD, best r):  0.7009056794330034
V-measure (SVD, best r):  0.6735674837344477
Adjusted Rand-Index (SVD, best r):  0.5946722914434844
Adjusted Mutual Information Score (SVD, best r):  0.672420502845868
```

[21]:
```
svd = TruncatedSVD(n_components=best_r_SVD, random_state=0)
svd_km = svd.fit_transform(data_feat)
kmean_svd = km.fit(svd_km)
cm = confusion_matrix(y_GT, kmean_svd.labels_)
rows, cols = linear_sum_assignment(cm, maximize=True)
plot_mat(cm[rows[:, np.newaxis], cols], xticklabels=cols,␣
 ↪yticklabels=rows,title = 'SVD (r = 50), K-Means', size=(6,4),pic_fname =␣
 ↪'Q167.png')

nmf_normal = NMF(n_components=best_r_NMF,init='random', random_state=1,␣
 ↪max_iter=1000)
nmf_km_normal = nmf_normal.fit_transform(data_feat)
kmean_nmf_normal= km.fit(nmf_km_normal)
cm = confusion_matrix(y_GT, kmean_nmf_normal.labels_)
rows, cols = linear_sum_assignment(cm, maximize=True)
plot_mat(cm[rows[:, np.newaxis], cols], xticklabels=cols,␣
 ↪yticklabels=rows,title = 'NMF (r = 5), K-Means', size=(6,4),pic_fname =␣
 ↪'Q168.png')


Umap_euc = umap.UMAP(n_components=best_r_euc, metric='euclidean').
 ↪fit_transform(data_feat)
kmean_euc = km.fit(Umap_euc)
cm = confusion_matrix(y_GT, kmean_euc.labels_)
rows, cols = linear_sum_assignment(cm, maximize=True)
plot_mat(cm[rows[:, np.newaxis], cols], xticklabels=cols, yticklabels=rows,␣
 ↪title = 'Euclidean UMAP (r = 100), K-Means', size=(6,4),pic_fname = 'Q169.
 ↪png')

Umap_cos = umap.UMAP(n_components=best_r_cos, metric='cosine').
 ↪fit_transform(data_feat)
kmean_cos = km.fit(Umap_cos)
cm = confusion_matrix(y_GT, kmean_cos.labels_)
rows, cols = linear_sum_assignment(cm, maximize=True)
plot_mat(cm[rows[:, np.newaxis], cols], xticklabels=cols, yticklabels=rows,␣
 ↪title = 'Cosine UMAP (r = 5), K-Means', size=(6,4),pic_fname = 'Q1610.png')
```


SVD (r = 50), K-Means
0
1
2
3
4
328 8 0 0 0
10 246 0 3 2
121 6 146 1 0
20 0 0 326 0
20 134 0 2 117
0 2 4 1 3
300
250
200
150
100
50
0


NMF (r = 5), K-Means
0
1
2
3
4
330 5 0 1 0
29 222 0 7 3
115 1 158 0 0
19 0 0 326 1
138 9 0 1 125
3 0 2 4 1
300
250
200
150
100
50
0


Euclidean UMAP (r = 100), K-Means
0 296 8 22 7 3 0
1 10 241 5 0 5 1
2 19 4 246 2 3 2
3 3 0 1 342 0 3
4 4 20 9 2 238 4
2 0 4 1 3
300
250
200
150
100
50
0


Cosine UMAP (r = 5), K-Means
0 295 8 22 8 3 0
1 10 242 5 0 4 1
2 18 4 247 2 3 2
3 3 0 1 342 0 3
4 4 15 9 2 243 4
3 1 0 2 4
300
250
200
150
100
50
0

b. Agglomerative clustering, DBSCAN and HDBSCAN with UMAP (Cosine)

[44]:
```
Umap_cos = umap.UMAP(n_components=10, metric='cosine').fit_transform(data_feat)
ac_w = AgglomerativeClustering(n_clusters=5, linkage='ward').fit(Umap_cos)
ac_s = AgglomerativeClustering(n_clusters=5, linkage='single').fit(Umap_cos)
```

[45]:
```
eps_rec = []
min_samples_rec = []
db_hs = []
db_cs = []
db_vs = []
db_ari = []
db_ms = []

eps = [0.01,0.05, 0.1, 0.15, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0,3.0,5.
 ↪0,10.0,30.0,50.0]
min_samples = [5,15,30,60,100,200,500,1000,3000]

for i in range(len(eps)):
    for j in range(len(min_samples)):
        print('Testing for e and min_sample = ',eps[i],'and',min_samples[j])
        dbs = DBSCAN(eps=eps[i],min_samples=min_samples[j],n_jobs=-1).
 ↪fit_predict(Umap_cos)
        db_hs.append(homogeneity_score(y_GT, dbs))
        db_cs.append(completeness_score(y_GT, dbs))
        db_vs.append(v_measure_score(y_GT, dbs))
        db_ari.append(adjusted_rand_score(y_GT, dbs))
        db_ms.append(adjusted_mutual_info_score(y_GT, dbs))
        eps_rec.append(eps[i])
        min_samples_rec.append(min_samples[j])
print('Done testing')
```

```
Testing for e and min_sample =  0.01 and 5
Testing for e and min_sample =  0.01 and 15
Testing for e and min_sample =  0.01 and 30
Testing for e and min_sample =  0.01 and 60
Testing for e and min_sample =  0.01 and 100
Testing for e and min_sample =  0.01 and 200
Testing for e and min_sample =  0.01 and 500
Testing for e and min_sample =  0.01 and 1000
Testing for e and min_sample =  0.01 and 3000
Testing for e and min_sample =  0.05 and 5
Testing for e and min_sample =  0.05 and 15
Testing for e and min_sample =  0.05 and 30
Testing for e and min_sample =  0.05 and 60
Testing for e and min_sample =  0.05 and 100
Testing for e and min_sample =  0.05 and 200
Testing for e and min_sample =  0.05 and 500
```

```
Testing for e and min_sample =  0.05 and 1000
Testing for e and min_sample =  0.05 and 3000
Testing for e and min_sample =  0.1 and 5
Testing for e and min_sample =  0.1 and 15
Testing for e and min_sample =  0.1 and 30
Testing for e and min_sample =  0.1 and 60
Testing for e and min_sample =  0.1 and 100
Testing for e and min_sample =  0.1 and 200
Testing for e and min_sample =  0.1 and 500
Testing for e and min_sample =  0.1 and 1000
Testing for e and min_sample =  0.1 and 3000
Testing for e and min_sample =  0.15 and 5
Testing for e and min_sample =  0.15 and 15
Testing for e and min_sample =  0.15 and 30
Testing for e and min_sample =  0.15 and 60
Testing for e and min_sample =  0.15 and 100
Testing for e and min_sample =  0.15 and 200
Testing for e and min_sample =  0.15 and 500
Testing for e and min_sample =  0.15 and 1000
Testing for e and min_sample =  0.15 and 3000
Testing for e and min_sample =  0.2 and 5
Testing for e and min_sample =  0.2 and 15
Testing for e and min_sample =  0.2 and 30
Testing for e and min_sample =  0.2 and 60
Testing for e and min_sample =  0.2 and 100
Testing for e and min_sample =  0.2 and 200
Testing for e and min_sample =  0.2 and 500
Testing for e and min_sample =  0.2 and 1000
Testing for e and min_sample =  0.2 and 3000
Testing for e and min_sample =  0.3 and 5
Testing for e and min_sample =  0.3 and 15
Testing for e and min_sample =  0.3 and 30
Testing for e and min_sample =  0.3 and 60
Testing for e and min_sample =  0.3 and 100
Testing for e and min_sample =  0.3 and 200
Testing for e and min_sample =  0.3 and 500
Testing for e and min_sample =  0.3 and 1000
Testing for e and min_sample =  0.3 and 3000
Testing for e and min_sample =  0.4 and 5
Testing for e and min_sample =  0.4 and 15
Testing for e and min_sample =  0.4 and 30
Testing for e and min_sample =  0.4 and 60
Testing for e and min_sample =  0.4 and 100
Testing for e and min_sample =  0.4 and 200
Testing for e and min_sample =  0.4 and 500
Testing for e and min_sample =  0.4 and 1000
Testing for e and min_sample =  0.4 and 3000
Testing for e and min_sample =  0.5 and 5
```

```
Testing for e and min_sample =  0.5 and 15
Testing for e and min_sample =  0.5 and 30
Testing for e and min_sample =  0.5 and 60
Testing for e and min_sample =  0.5 and 100
Testing for e and min_sample =  0.5 and 200
Testing for e and min_sample =  0.5 and 500
Testing for e and min_sample =  0.5 and 1000
Testing for e and min_sample =  0.5 and 3000
Testing for e and min_sample =  0.6 and 5
Testing for e and min_sample =  0.6 and 15
Testing for e and min_sample =  0.6 and 30
Testing for e and min_sample =  0.6 and 60
Testing for e and min_sample =  0.6 and 100
Testing for e and min_sample =  0.6 and 200
Testing for e and min_sample =  0.6 and 500
Testing for e and min_sample =  0.6 and 1000
Testing for e and min_sample =  0.6 and 3000
Testing for e and min_sample =  0.7 and 5
Testing for e and min_sample =  0.7 and 15
Testing for e and min_sample =  0.7 and 30
Testing for e and min_sample =  0.7 and 60
Testing for e and min_sample =  0.7 and 100
Testing for e and min_sample =  0.7 and 200
Testing for e and min_sample =  0.7 and 500
Testing for e and min_sample =  0.7 and 1000
Testing for e and min_sample =  0.7 and 3000
Testing for e and min_sample =  0.8 and 5
Testing for e and min_sample =  0.8 and 15
Testing for e and min_sample =  0.8 and 30
Testing for e and min_sample =  0.8 and 60
Testing for e and min_sample =  0.8 and 100
Testing for e and min_sample =  0.8 and 200
Testing for e and min_sample =  0.8 and 500
Testing for e and min_sample =  0.8 and 1000
Testing for e and min_sample =  0.8 and 3000
Testing for e and min_sample =  0.9 and 5
Testing for e and min_sample =  0.9 and 15
Testing for e and min_sample =  0.9 and 30
Testing for e and min_sample =  0.9 and 60
Testing for e and min_sample =  0.9 and 100
Testing for e and min_sample =  0.9 and 200
Testing for e and min_sample =  0.9 and 500
Testing for e and min_sample =  0.9 and 1000
Testing for e and min_sample =  0.9 and 3000
Testing for e and min_sample =  1.0 and 5
Testing for e and min_sample =  1.0 and 15
Testing for e and min_sample =  1.0 and 30
Testing for e and min_sample =  1.0 and 60
```

```
Testing for e and min_sample =  1.0 and 100
Testing for e and min_sample =  1.0 and 200
Testing for e and min_sample =  1.0 and 500
Testing for e and min_sample =  1.0 and 1000
Testing for e and min_sample =  1.0 and 3000
Testing for e and min_sample =  3.0 and 5
Testing for e and min_sample =  3.0 and 15
Testing for e and min_sample =  3.0 and 30
Testing for e and min_sample =  3.0 and 60
Testing for e and min_sample =  3.0 and 100
Testing for e and min_sample =  3.0 and 200
Testing for e and min_sample =  3.0 and 500
Testing for e and min_sample =  3.0 and 1000
Testing for e and min_sample =  3.0 and 3000
Testing for e and min_sample =  5.0 and 5
Testing for e and min_sample =  5.0 and 15
Testing for e and min_sample =  5.0 and 30
Testing for e and min_sample =  5.0 and 60
Testing for e and min_sample =  5.0 and 100
Testing for e and min_sample =  5.0 and 200
Testing for e and min_sample =  5.0 and 500
Testing for e and min_sample =  5.0 and 1000
Testing for e and min_sample =  5.0 and 3000
Testing for e and min_sample =  10.0 and 5
Testing for e and min_sample =  10.0 and 15
Testing for e and min_sample =  10.0 and 30
Testing for e and min_sample =  10.0 and 60
Testing for e and min_sample =  10.0 and 100
Testing for e and min_sample =  10.0 and 200
Testing for e and min_sample =  10.0 and 500
Testing for e and min_sample =  10.0 and 1000
Testing for e and min_sample =  10.0 and 3000
Testing for e and min_sample =  30.0 and 5
Testing for e and min_sample =  30.0 and 15
Testing for e and min_sample =  30.0 and 30
Testing for e and min_sample =  30.0 and 60
Testing for e and min_sample =  30.0 and 100
Testing for e and min_sample =  30.0 and 200
Testing for e and min_sample =  30.0 and 500
Testing for e and min_sample =  30.0 and 1000
Testing for e and min_sample =  30.0 and 3000
Testing for e and min_sample =  50.0 and 5
Testing for e and min_sample =  50.0 and 15
Testing for e and min_sample =  50.0 and 30
Testing for e and min_sample =  50.0 and 60
Testing for e and min_sample =  50.0 and 100
Testing for e and min_sample =  50.0 and 200
Testing for e and min_sample =  50.0 and 500
```

```
Testing for e and min_sample =  50.0 and 1000
Testing for e and min_sample =  50.0 and 3000
Done testing
```

[46]:
```
hdb_hs = []
hdb_cs = []
hdb_vs = []
hdb_ari = []
hdb_ms = []

for i in range(len(eps)):
    for j in range(len(min_samples)):
        print('Testing for e and min_sample = ',eps[i],'and',min_samples[j])
        hdbs = hdbscan.
↪HDBSCAN(min_cluster_size=100,cluster_selection_epsilon=eps[i],min_samples=min_samples[j],co
↪fit_predict(Umap_cos)
        hdb_hs.append(homogeneity_score(y_GT, hdbs))
        hdb_cs.append(completeness_score(y_GT, hdbs))
        hdb_vs.append(v_measure_score(y_GT, hdbs))
        hdb_ari.append(adjusted_rand_score(y_GT, hdbs))
        hdb_ms.append(adjusted_mutual_info_score(y_GT, hdbs))
print('Done testing')
```

```
Testing for e and min_sample =  0.01 and 5
Testing for e and min_sample =  0.01 and 15
Testing for e and min_sample =  0.01 and 30
Testing for e and min_sample =  0.01 and 60
Testing for e and min_sample =  0.01 and 100
Testing for e and min_sample =  0.01 and 200
Testing for e and min_sample =  0.01 and 500
Testing for e and min_sample =  0.01 and 1000
Testing for e and min_sample =  0.01 and 3000
Testing for e and min_sample =  0.05 and 5
Testing for e and min_sample =  0.05 and 15
Testing for e and min_sample =  0.05 and 30
Testing for e and min_sample =  0.05 and 60
Testing for e and min_sample =  0.05 and 100
Testing for e and min_sample =  0.05 and 200
Testing for e and min_sample =  0.05 and 500
Testing for e and min_sample =  0.05 and 1000
Testing for e and min_sample =  0.05 and 3000
Testing for e and min_sample =  0.1 and 5
Testing for e and min_sample =  0.1 and 15
Testing for e and min_sample =  0.1 and 30
Testing for e and min_sample =  0.1 and 60
Testing for e and min_sample =  0.1 and 100
Testing for e and min_sample =  0.1 and 200
Testing for e and min_sample =  0.1 and 500
```

```
Testing for e and min_sample =  0.1 and 1000
Testing for e and min_sample =  0.1 and 3000
Testing for e and min_sample =  0.15 and 5
Testing for e and min_sample =  0.15 and 15
Testing for e and min_sample =  0.15 and 30
Testing for e and min_sample =  0.15 and 60
Testing for e and min_sample =  0.15 and 100
Testing for e and min_sample =  0.15 and 200
Testing for e and min_sample =  0.15 and 500
Testing for e and min_sample =  0.15 and 1000
Testing for e and min_sample =  0.15 and 3000
Testing for e and min_sample =  0.2 and 5
Testing for e and min_sample =  0.2 and 15
Testing for e and min_sample =  0.2 and 30
Testing for e and min_sample =  0.2 and 60
Testing for e and min_sample =  0.2 and 100
Testing for e and min_sample =  0.2 and 200
Testing for e and min_sample =  0.2 and 500
Testing for e and min_sample =  0.2 and 1000
Testing for e and min_sample =  0.2 and 3000
Testing for e and min_sample =  0.3 and 5
Testing for e and min_sample =  0.3 and 15
Testing for e and min_sample =  0.3 and 30
Testing for e and min_sample =  0.3 and 60
Testing for e and min_sample =  0.3 and 100
Testing for e and min_sample =  0.3 and 200
Testing for e and min_sample =  0.3 and 500
Testing for e and min_sample =  0.3 and 1000
Testing for e and min_sample =  0.3 and 3000
Testing for e and min_sample =  0.4 and 5
Testing for e and min_sample =  0.4 and 15
Testing for e and min_sample =  0.4 and 30
Testing for e and min_sample =  0.4 and 60
Testing for e and min_sample =  0.4 and 100
Testing for e and min_sample =  0.4 and 200
Testing for e and min_sample =  0.4 and 500
Testing for e and min_sample =  0.4 and 1000
Testing for e and min_sample =  0.4 and 3000
Testing for e and min_sample =  0.5 and 5
Testing for e and min_sample =  0.5 and 15
Testing for e and min_sample =  0.5 and 30
Testing for e and min_sample =  0.5 and 60
Testing for e and min_sample =  0.5 and 100
Testing for e and min_sample =  0.5 and 200
Testing for e and min_sample =  0.5 and 500
Testing for e and min_sample =  0.5 and 1000
Testing for e and min_sample =  0.5 and 3000
Testing for e and min_sample =  0.6 and 5
```

```
Testing for e and min_sample =  0.6 and 15
Testing for e and min_sample =  0.6 and 30
Testing for e and min_sample =  0.6 and 60
Testing for e and min_sample =  0.6 and 100
Testing for e and min_sample =  0.6 and 200
Testing for e and min_sample =  0.6 and 500
Testing for e and min_sample =  0.6 and 1000
Testing for e and min_sample =  0.6 and 3000
Testing for e and min_sample =  0.7 and 5
Testing for e and min_sample =  0.7 and 15
Testing for e and min_sample =  0.7 and 30
Testing for e and min_sample =  0.7 and 60
Testing for e and min_sample =  0.7 and 100
Testing for e and min_sample =  0.7 and 200
Testing for e and min_sample =  0.7 and 500
Testing for e and min_sample =  0.7 and 1000
Testing for e and min_sample =  0.7 and 3000
Testing for e and min_sample =  0.8 and 5
Testing for e and min_sample =  0.8 and 15
Testing for e and min_sample =  0.8 and 30
Testing for e and min_sample =  0.8 and 60
Testing for e and min_sample =  0.8 and 100
Testing for e and min_sample =  0.8 and 200
Testing for e and min_sample =  0.8 and 500
Testing for e and min_sample =  0.8 and 1000
Testing for e and min_sample =  0.8 and 3000
Testing for e and min_sample =  0.9 and 5
Testing for e and min_sample =  0.9 and 15
Testing for e and min_sample =  0.9 and 30
Testing for e and min_sample =  0.9 and 60
Testing for e and min_sample =  0.9 and 100
Testing for e and min_sample =  0.9 and 200
Testing for e and min_sample =  0.9 and 500
Testing for e and min_sample =  0.9 and 1000
Testing for e and min_sample =  0.9 and 3000
Testing for e and min_sample =  1.0 and 5
Testing for e and min_sample =  1.0 and 15
Testing for e and min_sample =  1.0 and 30
Testing for e and min_sample =  1.0 and 60
Testing for e and min_sample =  1.0 and 100
Testing for e and min_sample =  1.0 and 200
Testing for e and min_sample =  1.0 and 500
Testing for e and min_sample =  1.0 and 1000
Testing for e and min_sample =  1.0 and 3000
Testing for e and min_sample =  3.0 and 5
Testing for e and min_sample =  3.0 and 15
Testing for e and min_sample =  3.0 and 30
Testing for e and min_sample =  3.0 and 60
```

```
Testing for e and min_sample =  3.0 and 100
Testing for e and min_sample =  3.0 and 200
Testing for e and min_sample =  3.0 and 500
Testing for e and min_sample =  3.0 and 1000
Testing for e and min_sample =  3.0 and 3000
Testing for e and min_sample =  5.0 and 5
Testing for e and min_sample =  5.0 and 15
Testing for e and min_sample =  5.0 and 30
Testing for e and min_sample =  5.0 and 60
Testing for e and min_sample =  5.0 and 100
Testing for e and min_sample =  5.0 and 200
Testing for e and min_sample =  5.0 and 500
Testing for e and min_sample =  5.0 and 1000
Testing for e and min_sample =  5.0 and 3000
Testing for e and min_sample =  10.0 and 5
Testing for e and min_sample =  10.0 and 15
Testing for e and min_sample =  10.0 and 30
Testing for e and min_sample =  10.0 and 60
Testing for e and min_sample =  10.0 and 100
Testing for e and min_sample =  10.0 and 200
Testing for e and min_sample =  10.0 and 500
Testing for e and min_sample =  10.0 and 1000
Testing for e and min_sample =  10.0 and 3000
Testing for e and min_sample =  30.0 and 5
Testing for e and min_sample =  30.0 and 15
Testing for e and min_sample =  30.0 and 30
Testing for e and min_sample =  30.0 and 60
Testing for e and min_sample =  30.0 and 100
Testing for e and min_sample =  30.0 and 200
Testing for e and min_sample =  30.0 and 500
Testing for e and min_sample =  30.0 and 1000
Testing for e and min_sample =  30.0 and 3000
Testing for e and min_sample =  50.0 and 5
Testing for e and min_sample =  50.0 and 15
Testing for e and min_sample =  50.0 and 30
Testing for e and min_sample =  50.0 and 60
Testing for e and min_sample =  50.0 and 100
Testing for e and min_sample =  50.0 and 200
Testing for e and min_sample =  50.0 and 500
Testing for e and min_sample =  50.0 and 1000
Testing for e and min_sample =  50.0 and 3000
Done testing
```

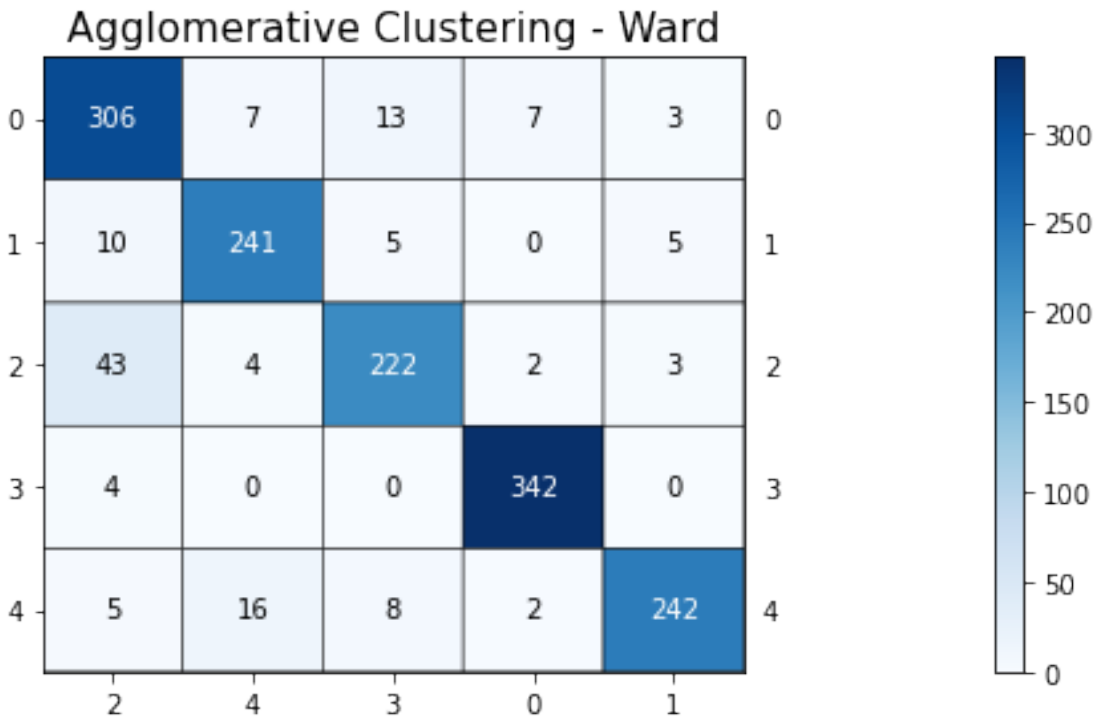
Performance Evaluation of Agglomerative clustering, DBSCAN and HDBSCAN

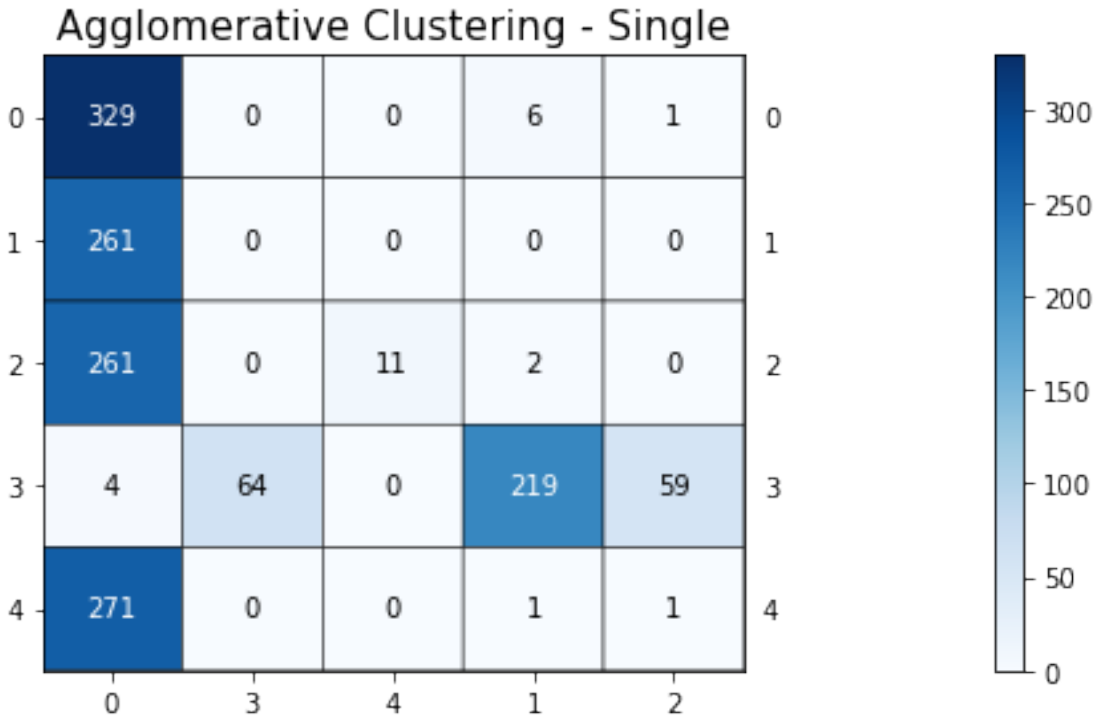
[47]:
```
print("Agglomerative Clustering, Ward - Homogeneity: %0.3f" %␣
 ↪homogeneity_score(y_GT, ac_w.labels_))
```

```
print("Agglomerative Clustering, Ward - Completeness: %0.3f" %␣
 ↪completeness_score(y_GT, ac_w.labels_))
print("Agglomerative Clustering, Ward - V-measure: %0.3f" %␣
 ↪v_measure_score(y_GT, ac_w.labels_))
print("Agglomerative Clustering, Ward - Adjusted Rand-Index: %.3f"%␣
 ↪adjusted_rand_score(y_GT, ac_w.labels_))
print("Agglomerative Clustering, Ward - Adjusted Mutual Information Score: %.
 ↪3f"% adjusted_mutual_info_score(y_GT, ac_w.labels_))
print("----------------------------------------------------------------------------------
print("Agglomerative Clustering, Single - Homogeneity: %0.3f" %␣
 ↪homogeneity_score(y_GT, ac_s.labels_))
print("Agglomerative Clustering, Single - Completeness: %0.3f" %␣
 ↪completeness_score(y_GT, ac_s.labels_))
print("Agglomerative Clustering, Single - V-measure: %0.3f" %␣
 ↪v_measure_score(y_GT, ac_s.labels_))
print("Agglomerative Clustering, Single - Adjusted Rand-Index: %.3f"%␣
 ↪adjusted_rand_score(y_GT, ac_s.labels_))
print("Agglomerative Clustering, Single - Adjusted Mutual Information Score: %.
 ↪3f"% adjusted_mutual_info_score(y_GT, ac_s.labels_))
```

```
Agglomerative Clustering, Ward - Homogeneity: 0.764
Agglomerative Clustering, Ward - Completeness: 0.768
Agglomerative Clustering, Ward - V-measure: 0.766
Agglomerative Clustering, Ward - Adjusted Rand-Index: 0.793
Agglomerative Clustering, Ward - Adjusted Mutual Information Score: 0.765
----------------------------------------------------------------------------------
----------------
Agglomerative Clustering, Single - Homogeneity: 0.316
Agglomerative Clustering, Single - Completeness: 0.633
Agglomerative Clustering, Single - V-measure: 0.422
Agglomerative Clustering, Single - Adjusted Rand-Index: 0.168
Agglomerative Clustering, Single - Adjusted Mutual Information Score: 0.419
```

[48]:
```
cm = confusion_matrix(y_GT, ac_w.labels_)
rows, cols = linear_sum_assignment(cm, maximize=True)
plot_mat(cm[rows[:, np.newaxis], cols], xticklabels=cols,␣
 ↪yticklabels=rows,title = 'Agglomerative Clustering - Ward',␣
 ↪size=(6,4),pic_fname = 'Q1611.png')

cm = confusion_matrix(y_GT, ac_s.labels_)
rows, cols = linear_sum_assignment(cm, maximize=True)
plot_mat(cm[rows[:, np.newaxis], cols], xticklabels=cols,␣
 ↪yticklabels=rows,title = 'Agglomerative Clustering - Single',␣
 ↪size=(6,4),pic_fname = 'Q1612.png')
```


Agglomerative Clustering - Ward
0
1
2
3
4
306
7
13
7
3
10
241
5
0
5
43
4
222
2
3
4
0
0
342
0
5
16
8
2
242
2
4
3
0
1
300
250
200
150
100
50
0

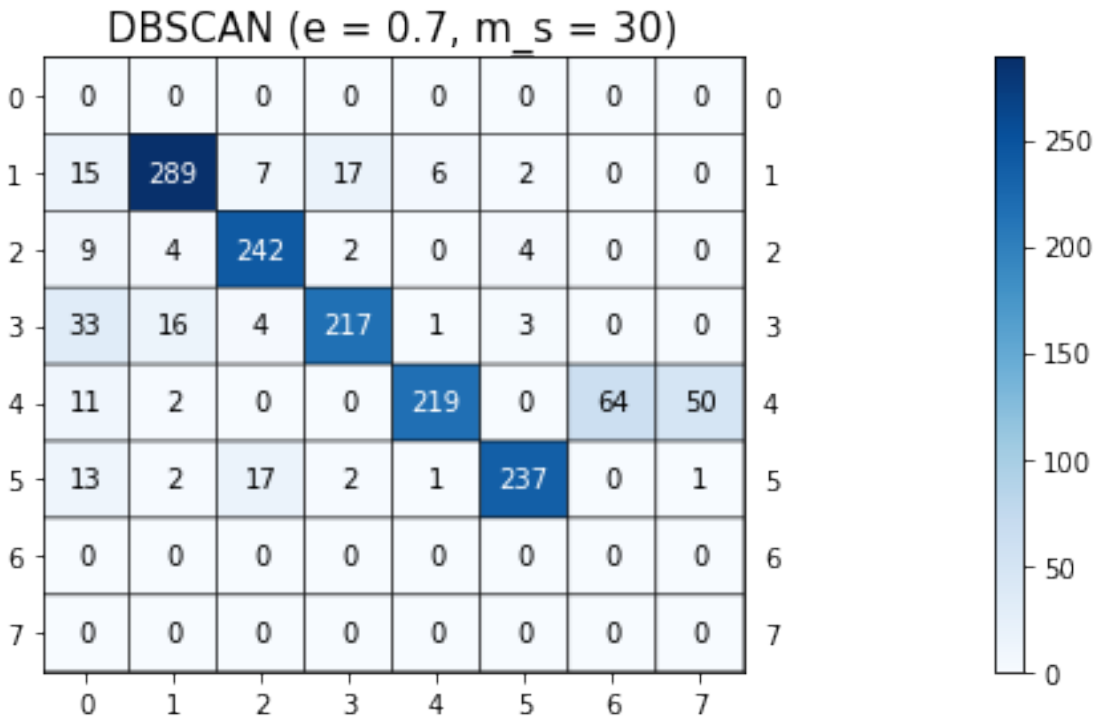

Agglomerative Clustering - Single
0
1
2
3
4
329
0
0
6
1
261
0
0
0
0
261
0
11
2
0
4
64
0
219
59
271
0
0
1
1
0
3
4
1
2
300
250
200
150
100
50
0

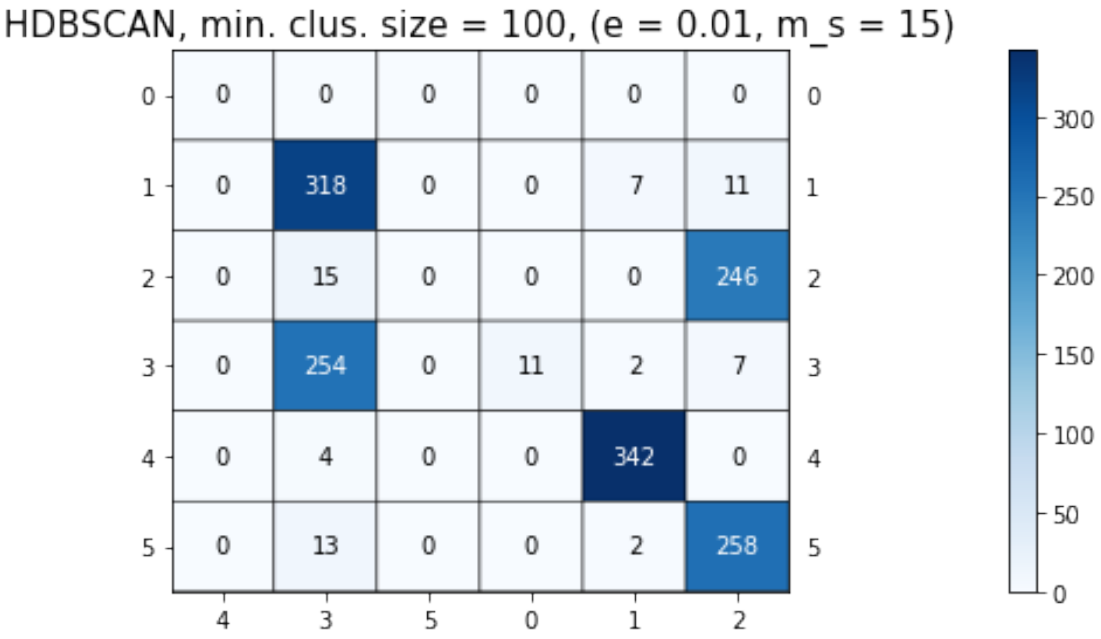
[49]:
```
avg_metrics = [y/5 for y in [sum(x) for x in zip(db_hs, db_cs, db_vs, db_ari,␣
 ↪db_ms)]]
best_eps_db = eps_rec[avg_metrics.index(max(avg_metrics))]
best_minSample_db = min_samples_rec[avg_metrics.index(max(avg_metrics))]
print('Best value of epsilon and minimum number of samples hyperparameters for␣
 ↪DBSCAN: ', best_eps_db,'and',best_minSample_db, ' respectively, avg. value␣
 ↪of 5 metrics: ',max(avg_metrics))
print('Metrics: ')
print('Homogeneity (DBSCAN, best hyperparameters): ',db_hs[avg_metrics.
 ↪index(max(avg_metrics))])
print('Completeness (DBSCAN, best hyperparameters): ',db_cs[avg_metrics.
 ↪index(max(avg_metrics))])
print('V-measure (DBSCAN, best hyperparameters): ',db_vs[avg_metrics.
 ↪index(max(avg_metrics))])
print('Adjusted Rand-Index (DBSCAN, best hyperparameters): ',db_ari[avg_metrics.
 ↪index(max(avg_metrics))])
print('Adjusted Mutual Information Score (DBSCAN, best hyperparameters):␣
 ↪',db_ms[avg_metrics.index(max(avg_metrics))])
print("-------------------------------------------------------------------------------------
avg_metrics = [y/5 for y in [sum(x) for x in zip(hdb_hs, hdb_cs, hdb_vs,␣
 ↪hdb_ari, hdb_ms)]]
best_eps_hdb = eps_rec[avg_metrics.index(max(avg_metrics))]
best_minSample_hdb = min_samples_rec[avg_metrics.index(max(avg_metrics))]
print('Best value of epsilon and minimum number of samples hyperparameters for␣
 ↪HDBSCAN: ', best_eps_hdb,'and',best_minSample_hdb, ' respectively, avg.␣
 ↪value of 5 metrics: ',max(avg_metrics))
print('Metrics: ')
print('Homogeneity (HDBSCAN, best hyperparameters): ',hdb_hs[avg_metrics.
 ↪index(max(avg_metrics))])
print('Completeness (HDBSCAN, best hyperparameters): ',hdb_cs[avg_metrics.
 ↪index(max(avg_metrics))])
print('V-measure (HDBSCAN, best hyperparameters): ',hdb_vs[avg_metrics.
 ↪index(max(avg_metrics))])
print('Adjusted Rand-Index (HDBSCAN, best hyperparameters):␣
 ↪',hdb_ari[avg_metrics.index(max(avg_metrics))])
print('Adjusted Mutual Information Score (HDBSCAN, best hyperparameters):␣
 ↪',hdb_ms[avg_metrics.index(max(avg_metrics))])
```

```
Best value of epsilon and minimum number of samples hyperparameters for DBSCAN:
0.7 and 30  respectively, avg. value of 5 metrics:  0.7100000647110825
Metrics:
Homogeneity (DBSCAN, best hyperparameters):  0.7812601834858268
Completeness (DBSCAN, best hyperparameters):  0.6508983166825818
V-measure (DBSCAN, best hyperparameters):  0.7101461720364782
Adjusted Rand-Index (DBSCAN, best hyperparameters):  0.6991186381134069
Adjusted Mutual Information Score (DBSCAN, best hyperparameters):
```

```
0.708577013237118
--------------------------------------------------------------------------------
---------------
Best value of epsilon and minimum number of samples hyperparameters for HDBSCAN:
0.01 and 15  respectively, avg. value of 5 metrics:  0.653956470738123
Metrics:
Homogeneity (HDBSCAN, best hyperparameters):  0.5631797605582871
Completeness (HDBSCAN, best hyperparameters):  0.8104923940615928
V-measure (HDBSCAN, best hyperparameters):  0.6645732912133311
Adjusted Rand-Index (HDBSCAN, best hyperparameters):  0.5680108102696699
Adjusted Mutual Information Score (HDBSCAN, best hyperparameters):
0.6635260975877341
```

[52]:
```
dbs = DBSCAN(eps=best_eps_db,min_samples=best_minSample_db,n_jobs=-1).
 ↪fit_predict(Umap_cos)
cm = confusion_matrix(y_GT, dbs)
rows, cols = linear_sum_assignment(cm, maximize=True)
plot_mat(cm[rows[:, np.newaxis], cols], xticklabels=cols, yticklabels=rows,␣
 ↪title = 'DBSCAN (e = 0.7, m_s = 30)', size=(6,4),pic_fname = 'Q1613.png')

hdbs = hdbs = hdbscan.
 ↪HDBSCAN(min_cluster_size=100,cluster_selection_epsilon=best_eps_hdb,min_samples=best_minSam
 ↪fit_predict(Umap_cos)
cm = confusion_matrix(y_GT, hdbs)
rows, cols = linear_sum_assignment(cm, maximize=True)
plot_mat(cm[rows[:, np.newaxis], cols], xticklabels=cols, yticklabels=rows,␣
 ↪title = 'HDBSCAN, min. clus. size = 100, (e = 0.01, m_s = 15)',␣
 ↪size=(6,4),pic_fname = 'Q1614.png')
```


DBSCAN (e = 0.7, m_s = 30)
0 0 0 0 0 0 0 0 0
1 15 289 7 17 6 2 0 0 1
2 9 4 242 2 0 4 0 0 2
3 33 16 4 217 1 3 0 0 3
4 11 2 0 0 219 0 64 50 4
5 13 2 17 2 1 237 0 1 5
6 0 0 0 0 0 0 0 0 6
7 0 0 0 0 0 0 0 0 7
0 1 2 3 4 5 6 7
250
200
150
100
50
0


HDBSCAN, min. clus. size = 100, (e = 0.01, m_s = 15)
0 0 0 0 0 0 0 0
1 0 318 0 0 7 11 1
2 0 15 0 0 0 246 2
3 0 254 0 11 2 7 3
4 0 4 0 0 342 0 4
5 0 13 0 0 2 258 5
4 3 5 0 1 2
300
250
200
150
100
50
0

[ ]: