SAN JOSE STATE UNIVERSITY

DEPARTMENT OF MATHEMATICS AND STATISTICS

Adaptive Spectral Clustering for High-Dimensional Sparse Count Data

Author:
Team Leaders:
Joey Fitch, Fengmei Liu
Team Members:
Shiou-Shiou Deng,
Sonia Kong, Nate Kotila,
Rachel Li, Ryan Quigley,
Andrew Zastovnik

Faculty Supervisor:
Dr. Guangliang CHEN

A report submitted in fulfillment of the requirements
for
Applied Mathematics, Computing & Statistics Projects(CAMCOS)

in the

Spring 2017 CAMCOS Verizon Project

June 15, 2017

Abstract

Industry companies have large amount of data to get insights from, and the insights can provide valuable information so as to help companies to make next-step actions. In this project, we helped Verizon Wireless on their cellphone user data clustering by using the spectral clustering technique. An adaptive spectral clustering process was built and tested, which includes data processing, dimensionality reduction, similarity and clustering. Three different spectral clustering methods were implemented. In the end, insights were extracted from the clustering results. The process was tested on the 20 news group data. High clustering accuracies were achieved on several data subsets and the full data. This adaptive spectral clustering could be applied in many areas like document clustering and web user clustering.

Acknowledgements

We specially thank Prof. Chen and Prof. Simic for the project opportunity and guidance.

We thank Verizon for sponsoring this project, and thank Irina Pragin, Yong Liu, Santanu Das, and Debasish Das for their valuable time in hosting our visits and answering our many questions.

Contents

Chapter 1

Introduction

1.1 Project Introducion

As a wireless service provider, Verizon has a large amount of data about cellphone users, including users' demographic information and web browsing history. As illustrated in Figure 1.1 left, they would like to discover insights from these data. The interested topics contain customer segmentation, users feature prediction, and also the temporal variations of these insights, etc.

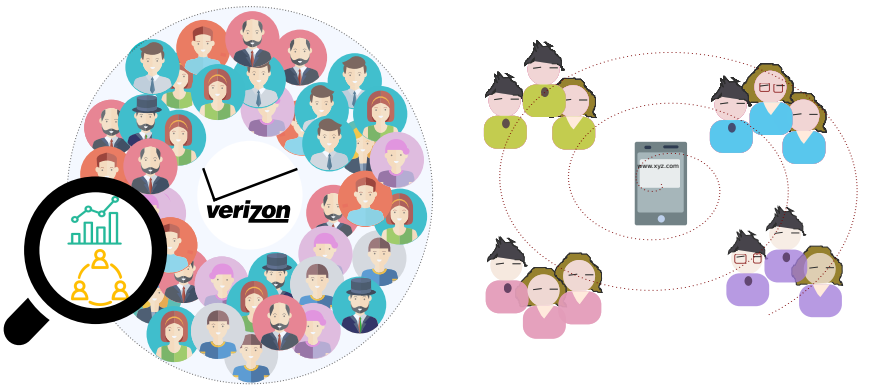


FIGURE 1.1: Insights from large user data

In this Spring 2017 CAMCOS project, we focused on customer segmentation, that is, to discover similar users based on their website browsing history, shown in Figure 1.1 right. From the statistical point of view, we regard this problem as a clustering problem, to group objects(users) based on their features(website). The clustering result will provide valuable information to Verizon and their business customers. Their marketing schemes can be built on the clustering results.

Clustering is a statistical data analysis technique, and belongs to the unsupervised machine learning field. The objective is to group similar objects while separating dissimilar objects. The fundamental problem of clustering is to find a proper way to measure proximity, including distances, similarities etc[6].

1.2 Project Data Format and Structure

The true data could not be made available to us because of the data confidentiality. Instead, we obtained a simulated dataset from Verizon, which mimics their true data structure and distribution very well.

Verizon gave us 71 days' user data information. Figure 1.2 shows us the structure of the data folders and dimension of each folder. Each user has around 70 variables including demographic and web browsing information. The "hist" folder contains all the variables and information we need. There are totally around 1 million users in the simulated dataset.

Original data are parquet files. The data extraction and initial exploration are done in Pyspark2 [11].

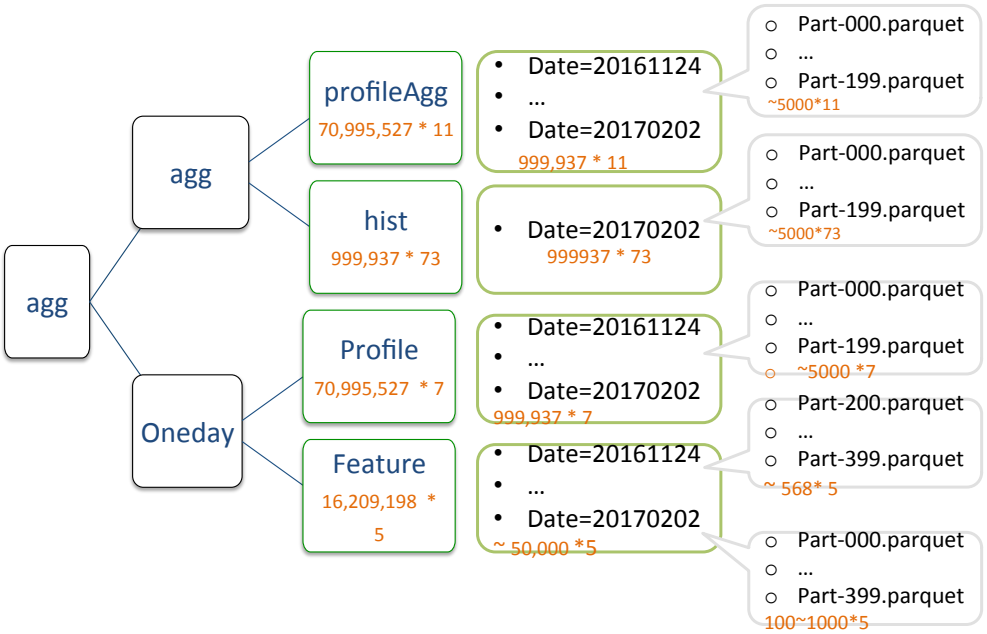


FIGURE 1.2: Verizon Simulated Data

As shown in Figure 1.3, we only took the aggregated website browsing data in the past 71 days, and transformed them to sparse matrix. After filtering out the "none" records in the column of "tldAggScore", we got about 330k users with 175k websites.

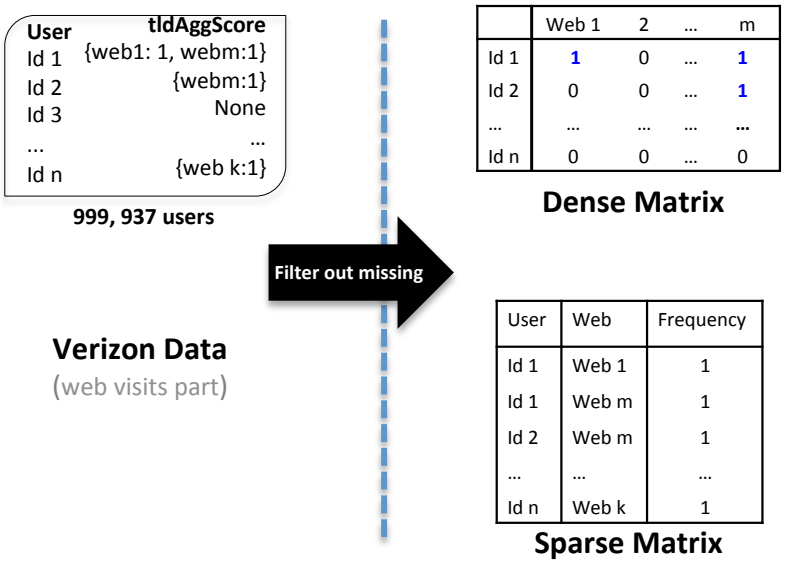


FIGURE 1.3: Extracted Data Matrix From Simulated Data

Two main features of this data are :

- High dimensional
  The dimension of the feature space is 175K.
- High sparsity

$$\text{Percentage of non-zero entries} = \frac{591K}{330K * 175K} = 0.001\% \tag{1.1}$$

Figure 1.4 shows a 200 user sample data. Dark points represent zero entries, and lighter points represent non-zero ones.

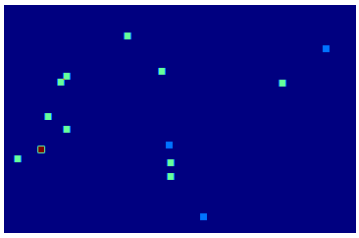


FIGURE 1.4: 200 Users Sample Data Matrix

1.3 Proof of Concept Research

The data true labels are unknown, so we are limited with the measures of the clustering methods and results. In this research, we conducted a proof of concept study, by using a classical dataset - "20 news group" dataset [8], which will be introduced in the next chapter. We implemented the process and tested it mainly using the 20 news group data. The results are compared and improved. Later we tried the process on the simulated data and get the insights from the data. In the end phase, Verizon

also implemented the function blocks on the true dataset. The whole process was developed and tested in R [14].

1.4 Overall Process Flow

The main process we developed for the data clustering is shown below in Figure 1.5. Starting with the data, the first step is data processing, which mainly includes column processing and row processing. Since the data is large, we did dimensionality reduction in the second step. This step is not a required step but it gives better result after test. The next process is similarity calculation, in which multiple methods are considered. The clustering is done based on similarity matrix. We explored multiple methods like Kmeans [5], Reduced Kmeans [3] etc, and finalized and focused with spectral clustering [16, 2]. The final step is getting insights from the clustering result, which is the most interesting part to industry customers.

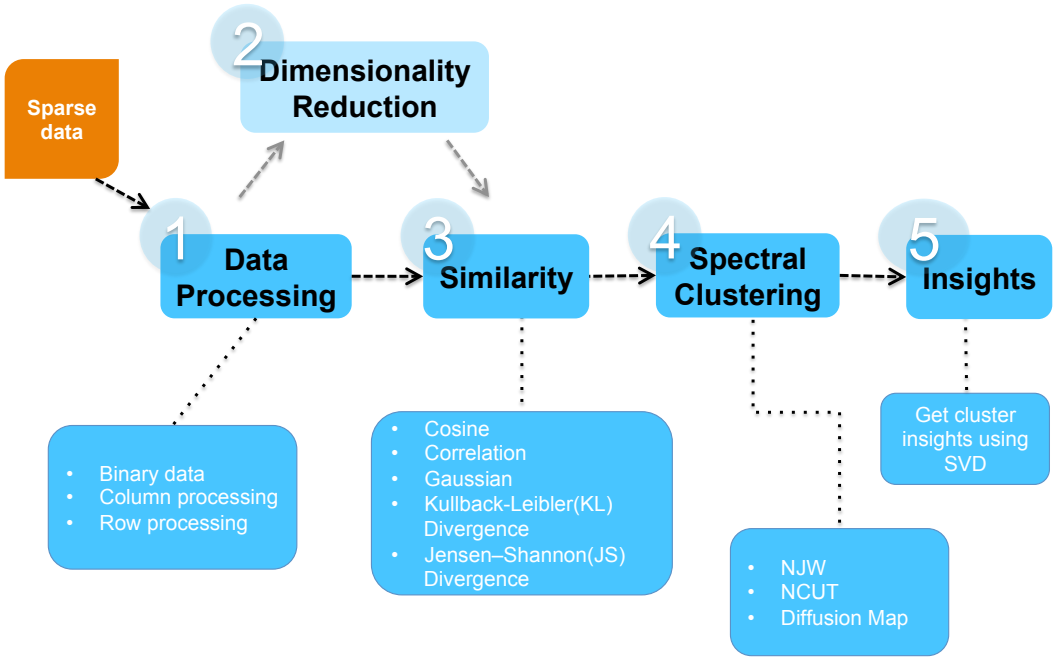


FIGURE 1.5: Clustering Process Flow

1.5 Report Organization

The rest of the report is organized as follows.

Chapter 2 will cover the 20 news group data exploration and summary. In Chapter 3 we will talk about the data processing we did to the columns and rows. Dimension reduction techniques will be shown in Chapter 4 and Chapter 5 is about the similarity measures we tried. Chapter 6 is to demonstrate the outlier-removal tests we tried.

Spectral Clustering techniques will be covered in Chapter 7. And, we will show our clustering insights and results in Chapter 8 and 9 separately. The ending chapter is the summary and proposed future work.

Chapter 2

20 News Group Data

2.1 Data Exploration

As introduced in the first chapter, the main challenges for Verizon data are its properties of high dimension and high sparsity. Since we do not have any information on the structure of the data, it is difficult to make the cluster analysis on it. Our strategy is to find a classical dataset that is similar to Verizon dataset and work on it to develop and test our algorithms. Once the methods and algorithms are finalized, we will apply them to the Verizon data. In this project we choose 20 News Group dataset (http://qwone.com/ jason/20Newsgroups/) as our emulated data set to work on. It is not our target data set; we use it as the proof of concept.

The 20 News Group dataset is an open resource from Internet.It comprises around 18,000 newsgroups documents on 20 topics with all labels available. It has been split in two subsets: one for training and the other one for testing. The split between the train and test set is based upon messages posted before and after a specific date. It is originally used as classification but in our project we did not use the labels for training purpose. We develop our algorithms without labels. We only use labels as ground truth to evaluate the performance of our methods. And also we use the training dataset only. Whenever we discuss the 20 News Group dataset in the rest of the report, we refer to its training set.

The total documents in the training set of the 20 newsgroup dataset are 11,269 which contain 53,975 unique words(including stop words). We did not remove any stop word. The density of the dataset is 0.0024. Fig.2.1 shows the counts for each word in the dataset. Most words appear at very low rate which is less than 5. Fig.2.2 shows how many words are there in each documents. Most documents contain around 100 words.

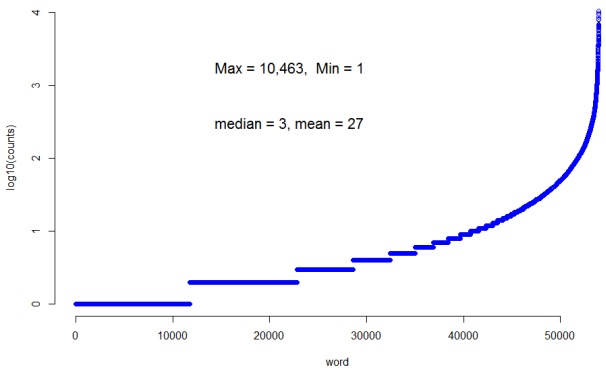


FIGURE 2.1: Word Counts(logarithmic scale)

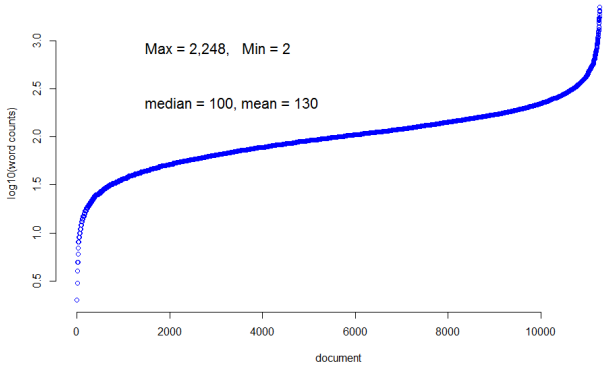


FIGURE 2.2: Words per Document(logarithmic scale)

2.2 Dataset Structure

Fig.2.3 shows the overview of the 20 News Group dataset. These 20 news topics belong to 6 categories(different colored in Fig.2.3), including comp(computer), politics, sci(science), rec(recreation), religion and misc(miscellaneous). In each category, there are several similar topics.For example, in the category of comp(computer),there are 5 topics including graphics,operation-system of Windows,IBM pc hardware, Mac hardware and windows.x. They are more related compared with the groups from different categories. It is more challenging to cluster on these groups from the same category. In our project, we will select different combinations of groups to test our algorithms.

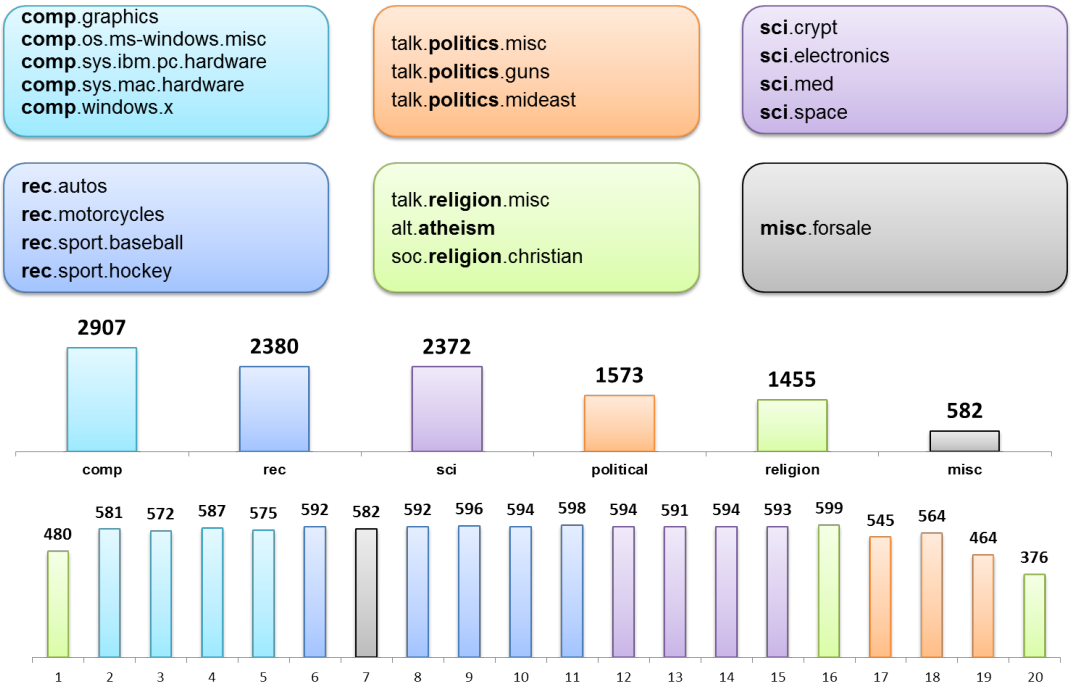


FIGURE 2.3: Overview of 20 News Group Dataset

2.3 Combinations of Groups

To test our proof of concept implementation, we develop six different tasks for 20 News Group dataset to recreate the results for sanity check. 20 News Group dataset which was introduced in the previous chapter could be separated to six categories. The combinations of groups are shown in table 2.1.

The first and the fourth tasks are selecting one small group from different categories. One is 3 clusters, and the other is 6 clusters, respectively. The former clusters are comp.graphics, rec.autos, and sci.crypt. The latter clusters are comp.graphics, rec.autos, sci.crypt, talk.politics.misc, talk.religion.misc, and misc.forsale.

The second and the third tasks are selecting all small groups form the same category. One is 4 clusters, REC category, and the other is 5 clusters, COMP category, respectively.

In the end, the fifth and the sixth tasks are using the full dataset. The former one is trying to separate into 6 clusters, and the latter one is trying to separate into 20 clusters.

Table 2.1: Combination of Groups for Sanity Check

task	Category	Group included
1	comp	comp.graphic
	rec	rec.autos
	sci	sci.crypt
2	rec	rec.autos
		rec.sport.baseball
		rec.sport.hockey
		rec.motorcycles
3	comp	comp.graphics,
		comp.os.ms-windows.misc,
		comp.sys.ibm.pc.hardware,
		comp.sys.mac.hardware,
		comp.windows.x
4	comp	comp.graphics
	rec	rec.autos
	sci	sci.crypt
	politics	talk.politics.misc
	religion	talk.religion.misc
	misc	misc.forsale
5	comp	full dataset
	rec	
	politics	
	sci	
	religion	
	misc	
6	comp	full dataset
	rec	
	politics	
	sci	
	religion	
	misc	

2.4 Form of the Dataset

We convert the documents collected in the 20 News Group dataset to word counts. That is, the appearance of each word in each document was counted and recorded. This generates a document term frequency matrix shown in table 2.2. We convert it to the sparse form which is shown in table 2.3. In the full matrix shown in table 2.2, the column represent words and row represent documents. It is a 11,269 * 53,975 matrix. The structure of 20 News Group data is pretty similar to that of the Verizon data. It also has same property as Verizon data including high dimension and very sparse structures. We can use it to mimic our target dataset and develop the algorithms.

Table 2.2: Full Matrix

	word 1	word 2	word 3	word 4	word 5	...	word n
doc 1	1	4	3	1	0	...	...
doc 2	9	1	0	2	0	...	...
doc 3	2	0	0	0	3	...	...
...	...	...	...	...	...	...	...
doc m	...	...	...	...	...	...	...

Table 2.3: Sparse Form

docID	wordID	count
doc 1	word 1	1
doc 1	word 2	4
doc 1	word 3	3
doc 1	word 4	1
doc 2	word 1	9
doc 2	word 2	1
doc 2	word 4	2
doc 3	word 1	2
doc 3	word 5	3
...	...	...
doc m	word n	...

Chapter 3

Data Processing

In this chapter we discuss the data processing methods tested and implemented in our algorithm. Many effective data processing techniques exist for text data such as the 20 newsgroups dataset; however, we tailored our methods to ensure that they could be generalized and applied to the Verizon data.

3.1 Binary

The first processing step is a decision about the format of the document term matrix. The original data contains word frequency per document; alternatively, all nonzero entries of the document term matrix can be converted to ones indicating that a word occurred in a document. In this alternative format no frequency information is retained, so there is loss of information as a result. The benefit is that words that tend to have a high frequency per document are de-emphasized, e.g. the, and, to, etc. Thus, converting all nonzero frequencies to ones gives all words the same weight within a particular document. This step consistently boosted clustering performance across all tests.

3.2 Column Processing

This section discusses the techniques applied columnwise to the document term matrix. These steps allow us makes changes across all documents to the influence of a particular word on the clustering results.

3.2.1 Column Trimming

Many of the words in the 20 news groups vocabulary are not useful features for clustering because they are at the extremes of document occurrence: too common or too rare. To handle this issue we use column trimming to reduce the number of columns to a more useful subset. This is accomplished by applying two thresholds to column sums. Since we have already converted the document term matrix to binary, the column sums represent the number of documents that a particular word occurs in. The two thresholds are:

1. Minimum document occurrence: 1
2. Maximum document occurrence: > 1000

The threshold values are chosen for specific reasons. The lower threshold removes any column corresponding to a word that only appeared in a single document. Since

we are attempting to cluster similar documents, a word occurring in only one document cannot be used to determine similarity between that document and others. Applying this threshold to the column sums removes approximately 9% of the columns in the document term matrix. This can be seen in the right tail of the plot in figure 3.1.

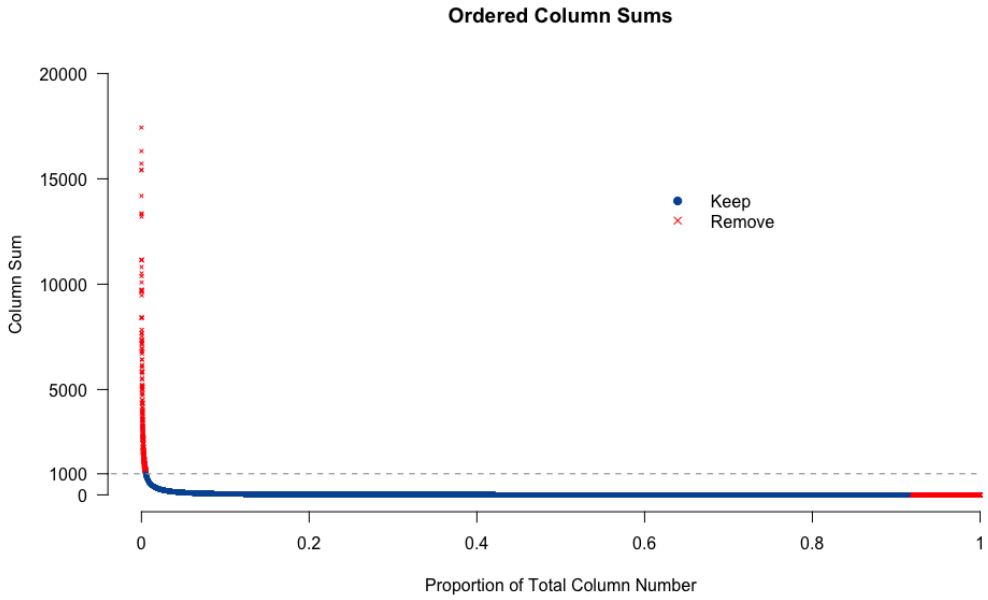


FIGURE 3.1: Column trimming thresholds applied to column sums

The upper threshold is used to remove words that are so prevalent that they do not contribute any useful information. In text analysis, these words are often removed by referring to list of a stop words; however, we do not want to remove words in this way because it would not generalize to the Verizon data. Thus for our purposes, the difficultly comes in choosing a value for this upper threshold. The value was chosen to correspond roughly to twice the average cluster size. The number of documents per cluster is summarized in figure 2.3. The reason being if a word is common enough that it occurs in nearly all documents of two separate clusters, then it will not provide useful information for clustering. This threshold removes approximately 0.5% of the columns, and this is illustrated in the left tail of the plot in figure 3.1.

3.2.2 Column Weighting

In an effort to further emphasize the most important words, we applied a weighting function to the columns. For an individual column, the weight is determined based on the column sum and is applied equally across all documents in that column. The functions we considered are:

- Step
- Linear
- Beta

- Inverse Document Frequency (IDF)
- IDF Squared

The motivation behind *step*, *linear*, and *beta* is similar to that from column trimming: we want to assign the least weight to the common or rare words. All three functions are essentially doing the same thing but with varying degrees of complexity and control. The general form of these functions can be seen in the first three plots of figure 3.2. *Beta* performed the best of these three functions, but there are a few disadvantages. Since we have already removed columns with a sum of one during column trimming, *beta* is heavily de-emphasizing words that only occur in a few documents; these words may be the key features indicating that those documents are indeed similar. The second disadvantage is that *beta* requires two additional parameters that must be tuned, and it can be quite difficult to choose two value that generalize well outside of the specific problem context. As a result, we turned to a different weighting function: inverse document frequency (IDF). IDF is parameter free and gives the highest weight to infrequent words. Mathematically it is defined,

$$\log\left(\frac{N}{n_t}\right)$$

where N is the total number of documents ad n_t is the column sum of the t-th column. The plot of the function can be seen in the bottom right panel of figure 3.2. IDF weighting led to better clustering results compared to all other weighting functions considered above. After seeing the success of IDF, we decided to take it one step further by squaring the weights output by the IDF function. This served to further emphasize the infrequent and unique words. The ratio of maximum weight to minimum weight from IDF was approximately 4, whereas the ratio from IDF squared was approximately 20. Squaring the IDF weights led to a noticeable boost in clustering accuracy, which will be discussed further in the results section.

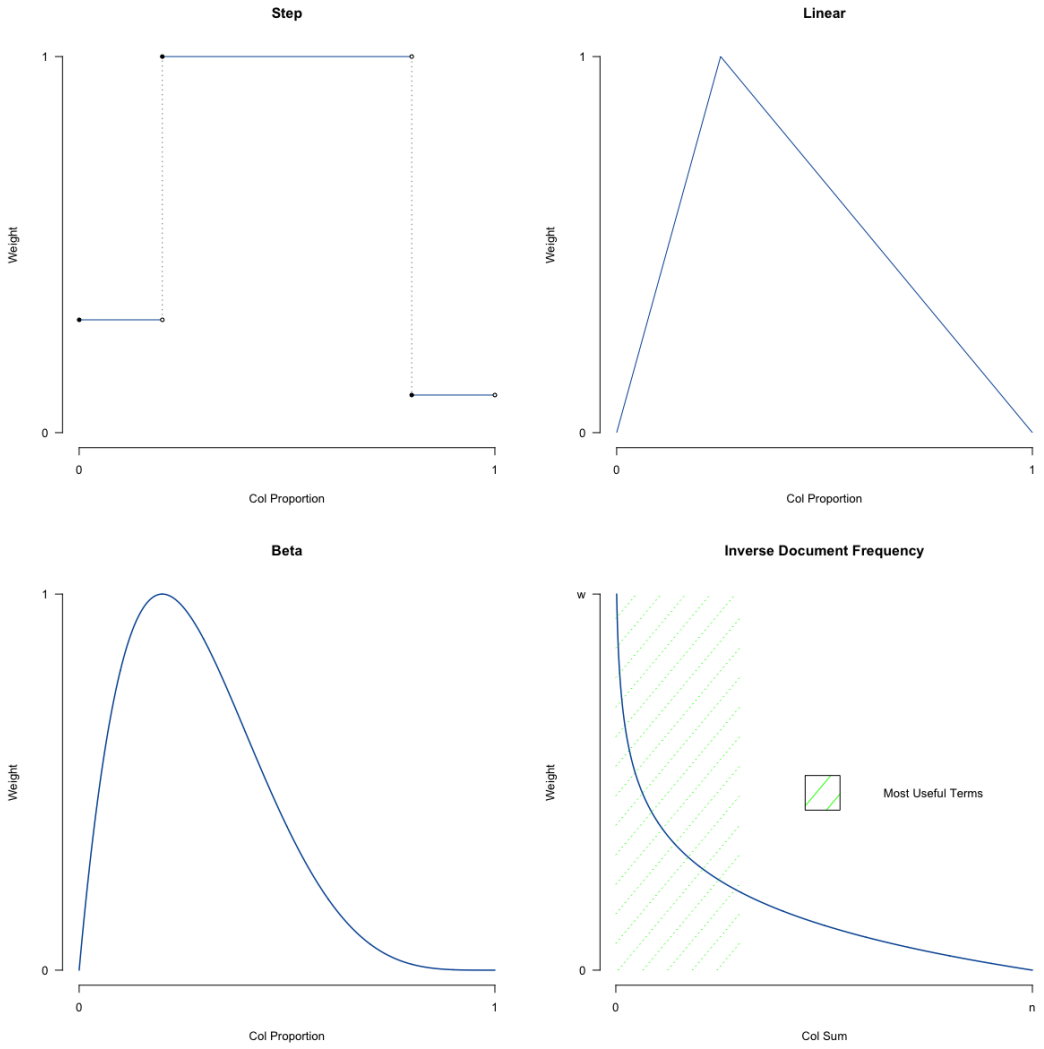


FIGURE 3.2: Column weighting functions

3.3 Row Processing

3.3.1 Row Trimming

After performing column trimming, it is possible that some documents (rows) will no longer have any nonzero entries. These rows prevent the necessary matrices operations from being performed later on during spectral clustering, so they are removed during this step. Intuitively, removing these rows makes sense because it is impossible to cluster the document without any information.

3.3.2 Row Normalization

The original documents from the 20 news groups data vary widely in length, so it is important to apply some form of row normalization to the documents in order to balance out the effect of document length. This step is performed after column weighting, so it is applied to the resulting weight matrix not the original document

term matrix. For the 20 news groups data, we considered L1 and L2 row normalization. For a given row, the two normalization methods are mathematically defined,

$$\text{L1} : \frac{w_j}{\sum_{j=1}^{p} |w_j|} \quad \text{L2} : \frac{w_j}{\sum_{j=1}^{p} w_j^2}$$

L1 normalization transforms the weights within each row into a discrete probability distribution as illustrated in the top panel of figure . L2 normalization projects each point onto the unit circle. In the bottom panel of figure , this is illustrated for two simulated clusters that are well separated in two dimensional space.

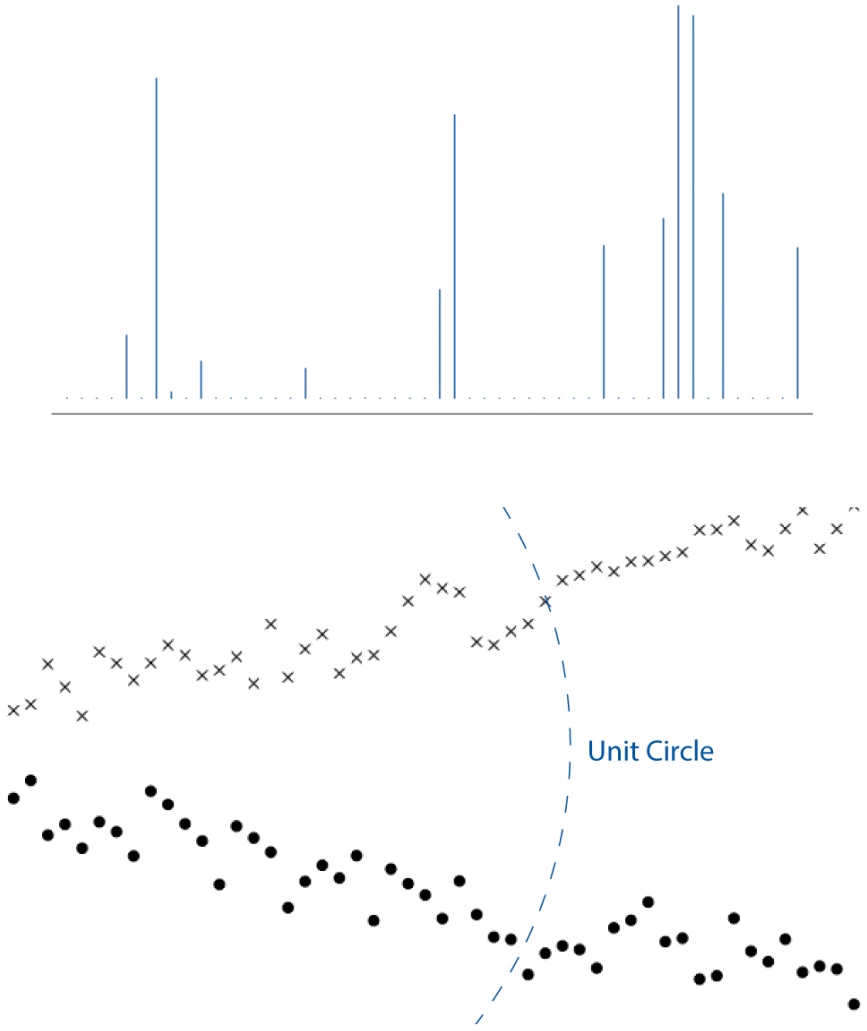


FIGURE 3.3: Illustration of L1 and L2 row normalization

In our tests, we found that normalization improved clustering results over no normalization and L2 outperformed L1. Thus, L2 normalization was applied as a data preprocessing step for all results displayed later on in this paper.

Chapter 4

Dimensionality Reduction

4.1 Motivation

Real data usually have thousands, or even millions of dimensions. Huge number of dimensions suffers from the so-called "curse of dimensionality", a phrase coined by ([1]) when considering problems in dynamic optimization. Another problem is that high dimensionality demands more memory for data storage and more time for data computation, which make many algorithms inefficient or even infeasible. The third problem is that data became very sparse, so that density based clustering algorithms become meaningless. Essentially, we assume that some of the data is noise, and we can approximate the useful part with a low dimensional part space. Dimensionality reduction not only reduces the dimension of data, but also suppress noise.

4.2 Latent Semantic Indexing

Latent Semantic Indexing (LSI) is a mathematical method used to perform a low-rank (say, k) approximation of document-term matrix (typical rank 100-300)([12]). The general idea is to design a mapping such that the low-dimensional space reflects semantic associations (latent semantic space), and then compute document similarity based on the inner product in this latent semantic space. Two goals of LSI are :

- Similar terms are mapped to nearby locations in low dimensional space
- Noise is reduced

4.3 Singular Value Decomposition

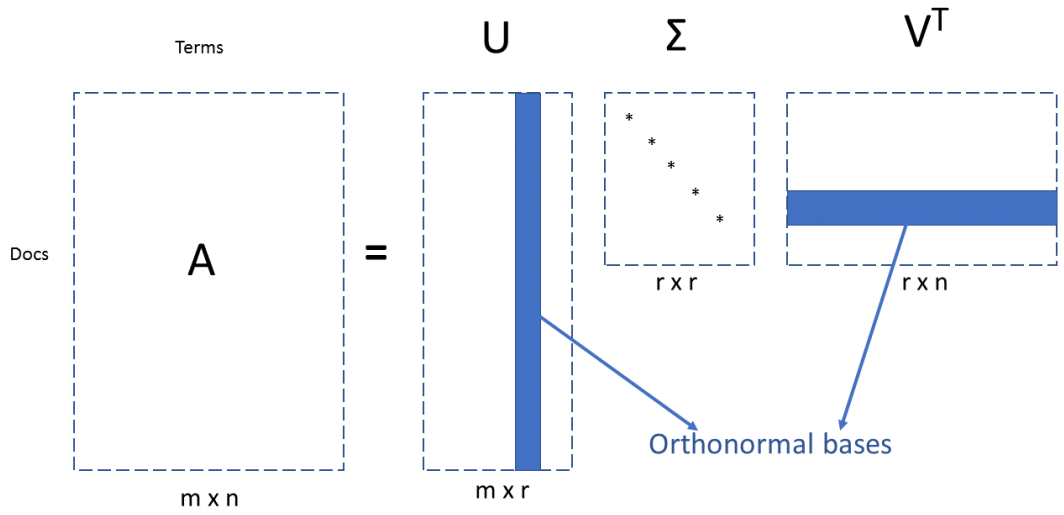


FIGURE 4.1: Singular Value Decomposition

Singular Value Decomposition (SVD) ([4]), is a mathematical way to implement the idea of LSI. Given a matrix A $\in IR^{m*n}$, The left singular vectors U are an orthonormal basis for the column space of A.The right singular vectors V are an orthonormal basis for the row space of A. The diagonal elements in Σ matrix are in descending order, which represent the strength of each subspace. If A has rank r, then A can be written as a sum of r rank-1 matrices. By keeping the top k strongest singular vectors, we map the original data into a top k dimensional subspace and obtain a reduced dataset B.

$$B = AV_k \tag{4.1}$$

In the nature language field, the k can range from around 100 to 300, in our case, we normally set k to a few hundred.

4.4 Illustration of 20newsgroup data with SVD

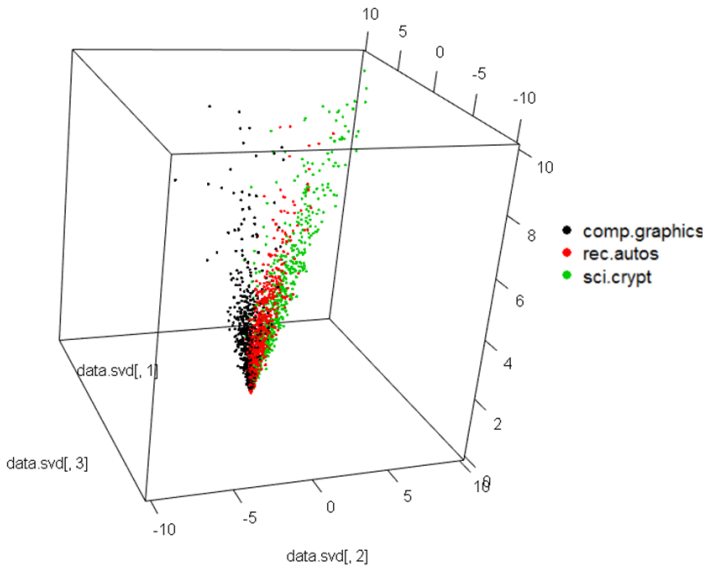


FIGURE 4.2: SVD-3 Clusters

In order to visualize data, we map our 20newsgroup dataset into k=3 dimensional semantic space through SVD, one point in the plot represents one document, the same color means the same group. We first picked up only 3 clusters, comp.graphics,rec.autos, and sci.crpyt as the Fig 4.2 shown, we can see clearly this reduced data presenting a significant structure that the 3 groups are well separated. If treat each document is a vector from origin, we can see the documents from the same group is heading to a specific direction, that may represent a specific topic. We can also see the documents in the same group have a small angle with each other, and a big angle with documents in the other groups.

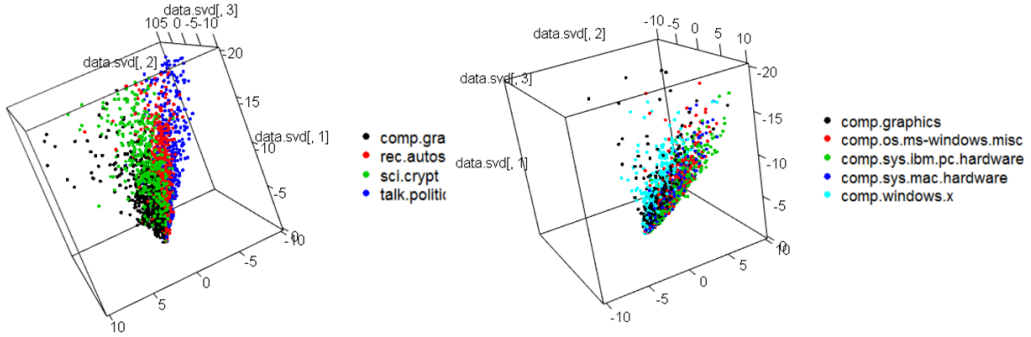


FIGURE 4.3: SVD-4 Clusters and 5 Clusters

In Fig 4.3, we picked the 4 clusters and 5 clusters from data. For the 4 clusters, we picked up groups comp.graph, rec.autos, sci.crypt, and talk.politics, these 4 groups are subgroups from the 4 big groups which are comp, rec, sci, and talk. since the topics are very different, we assume these 4 clusters are easier to distinguish, and hope to see a big angle from the plot. For the 5 clusters, we picked the groups comp.graphics, comp.os.ms-windows.misc, comp.sys.ibm.pc.hardware, comp.sys.mac.hardware, and comp.windows.x 6, which are from the same big group comp, we assume they are more difficult to distinguish since the similarity between each document are small.From the plot, we can see all the documents are heading to a same direction and the angle between each other are small.

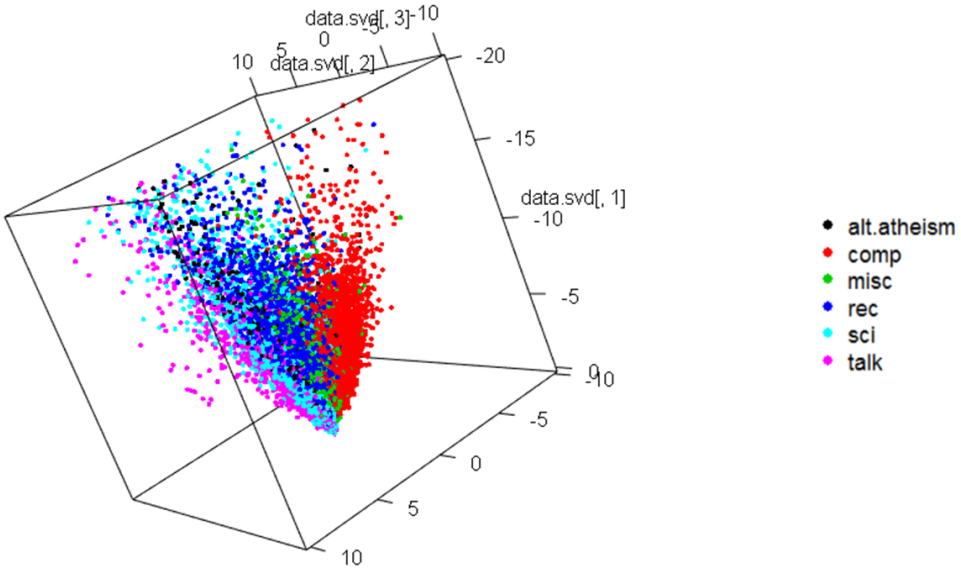


FIGURE 4.4: SVD-6 Clusters

At last, we picked all the dataset with 6 big clusters, alt.atheism,comp, misc.forsale, rec,sci, and talk. We can see from Fig 4.4, all the groups are well separated, so SVD is a feasible way for clustering in our case. It also provides us a way to consider the cosine similarity (angle) to measure the closeness of documents in the next step.

Chapter 5

Similarity

Once the data processing and dimensionality reduction steps are finished, our algorithm requires a similarity score between every pair of data observations. In the context of our datasets, we needed to compute pairwise similarity between every pair of documents or web users. Through this process, we transition from the original $n \times p$ data matrix into some $n \times n$ similarity matrix S, where the entry S_{ij} = Sim$(x_i, x_j) \in [0, 1]$. Intuitively, a strong similarity measure should be symmetric, reflexive, and nonnegative. In the following sections, we will discuss our strongest candidates for similarity metrics.

5.1 Gaussian Kernel Similarity

Our first idea was to calculate pairwise distances between all data observations, and then convert from distance to similarity using the Gaussian Kernel:

$$\text{Sim}(x, y) = e^{\frac{-\text{dist}(x,y)^2}{2\sigma^2}}$$

Given that distance is always nonnegative, the Gaussian Kernel is strictly $\in (0, 1]$, equal to 1 only if dist(x, y) = 0. In the usual Gaussian density function, σ^2 represents variance. Here, it similarly yields a scale parameter that specifies a "neighborhood" of similarity, allowing us to measure the data at different magnitudes of resolution.

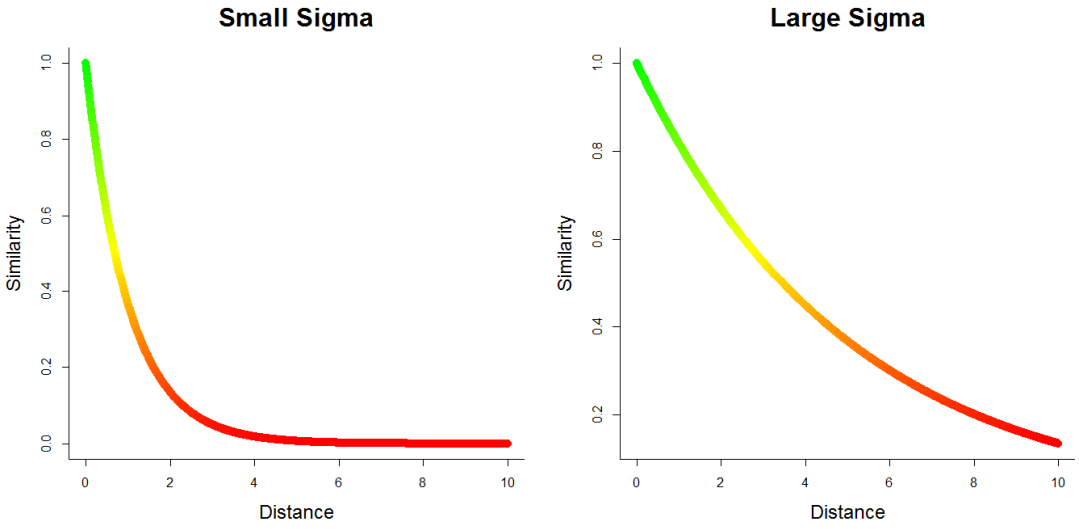


FIGURE 5.1: In the left figure, a small sigma only considers similarity between very small distances. In the right figure, a larger sigma increases the "neighborhood" of similarity to larger pairwise distances.

Beyond specifying a proper σ^2 value, we must also choose an adequate distance function. Any distance function can be inputted, bringing along all of its own pros and cons. We were intrigued by one particular distance function, known as Kullback–Leibler (KL) Divergence. This method treats each row as a discrete probability distribution and calculates the pairwise divergence between probability distributions (which are rows of data). This comes from information theory, where "divergence" of x and y is defined as the information loss when using distribution y to approximate distribution x. Similar observations should approximate each other well, yielding small divergence; dissimilar observations should approximate poorly, yielding large divergences. This can be expressed mathematically:

$$\text{KL-Div}(x, y) = \text{E}\left[\log(x) - \log(y)\right] = \sum_k x_k \left[\log\left(\frac{x_k}{y_k}\right)\right]$$

where x and y are *discrete probability distributions*. We convert frequency counts to discrete probability distributions by dividing each row by its row sum (the total frequency count for that observation).

The following figure shows an example of the algorithm applied between two data observations. The example uses full count data for clarity of demonstration, rather than binary data which would be more appropriate for our application. The algorithm remains the same in either case.

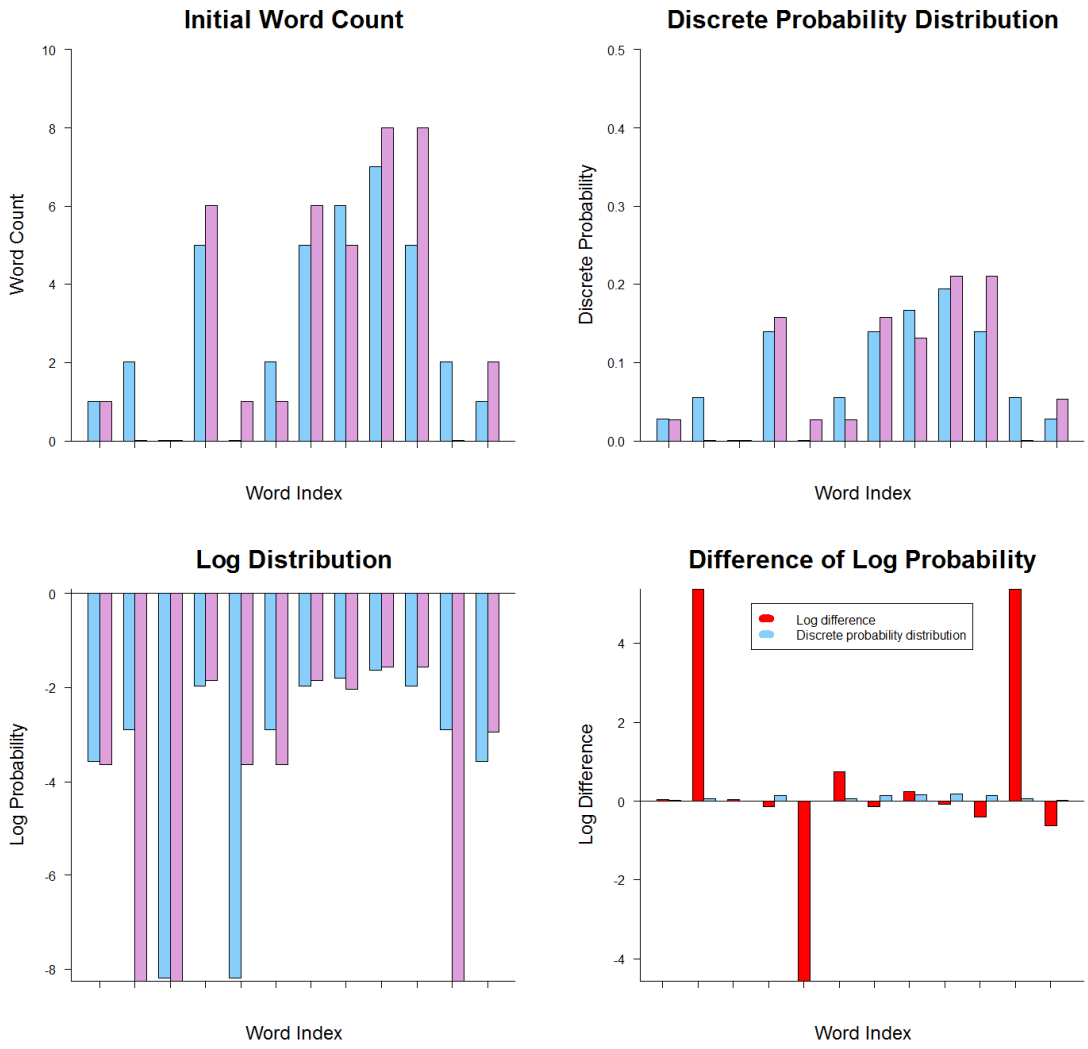


FIGURE 5.2: KL Divergence follows a 4–step algorithm to calculate distance between two observations (blue and purple):

1) We start with frequency counts for each column.
2) We convert frequency counts to discrete probability distributions by dividing the total frequency count for each document.
3) We take the log of each discrete probability value, going from [0,1] probability space to $(-\infty, 0)$ logarithm space.
4) We sum the difference in log probability for each column value.
5) KL Divergence is the sum of all log probability differences, weighted by the original discrete probabilities.

One important consequence arises from this formulation of divergence: the equation is not symmetric. Specifically, the expected value involves the probability distribution of x or the probability distribution of y, but it does not inherently reconcile any differences between them. We moved on to an extension of KL Divergence, known as Jensen–Shannon (JS) Divergence, which compares x and y to their average distribution $M = \frac{x+y}{2}$ rather than comparing x and y directly:

$$\text{JS-Div}(x, y) = \frac{\text{Divergence}(x,M) + \text{Divergence}(y,M)}{2}$$

$$\Rightarrow \text{Sim}(x, y) = e^{\frac{-\text{JS.Div}(x,y)^2}{2\sigma^2}} \qquad \text{(Gaussian Kernel)}$$

JS-Div is the average of KL-Div(x, M) and KL-Div(y, M), thereby achieving symmetry. However, we quickly reached a number of obstacles against full implementation of this algorithm. Most importantly: this process is computationally expensive. We never found an expedient way to calculate the full KL-Divergence matrix, much less the JS-Divergence matrix. This was a huge obstacle toward implementation on any practical dataset.

Even beyond the issues with JS Divergence specifically, the entire Gaussian Kernel method is dependent upon a couple problematic issues. To add to the problem of runtime, distance measures are generally always >0, eliminating the computational benefits of sparsity. When we tried running faster distance algorithms, the Gaussian Kernel estimate did not give successful results (relative to other similarity algorithms). Lastly, identifying the proper choice of σ^2 proved quite elusive. We tried strategies like k–nearest neighbor to specify sensible σ^2 values (trying many different k values), but we never discovered a method which gave strong results for our data. These methods for choosing σ^2 also compounded upon runtime issues, which were already a problem with most distance measures. We eventually decided to abandon the Gaussian Kernel entirely, in favor of other similarity methods.

5.2 Correlation

Instead of converting distance into similarity, we explored strategies to directly calculate similarity measures. One choice uses a variation of Pearson correlation to calculate pairwise similarity between rows of the data:

$$\text{Sim}(x, y) = \frac{\sum_k w_k(x_k - \mu_k)(y_k - \mu_k)}{\sqrt{\sum_k w_k(x_k - \mu_k)^2 \sum_k w_k(y_k - \mu_k)^2}} = \frac{\left(\vec{x_i} - \vec{\mu}\right)^T W \left(\vec{x_j} - \vec{\mu}\right)}{\left|\left|\vec{x_i} - \vec{\mu}\right|\right| \cdot \left|\left|\vec{x_j} - \vec{\mu}\right|\right|}$$

where ^{T} signifies the transpose for the matrix multiplication. Put more simply, we take the inner product of $\vec{x}$ and $\vec{y}$ after column centering and L2 row normalization.

This calculation essentially compares rows $\vec{x}$ and $\vec{y}$ according to their deviation from the mean in each column. If one or both rows have average values for a variable, that variable will contribute zero covariance to their overall correlation similarity. If these rows deviate from the mean in opposite directions, that variable will contribute negatively to their similarity. If these rows deviate from the mean in the same direction (whether above or below the mean), then that variable will contribute positively to their similarity. The stronger the mean deviations, the stronger the similarity contribution.

The fact that correlation similarity accounts for the mean in its calculation is actually very useful for us. In KL divergence, two large values in the same variable would contribute to low distance (high similarity). However, this variable may be

large for *all* the observations, meaning that a pair of large values is not particularly special there. For example, it is probably common for two documents to use the word "the" many times, or for two web users to visit the website Facebook often. This does not necessarily indicate similarity, because all different sorts of documents use the word "the", and many different types of web users all visit Facebook. Correlation similarity measures behavior *relative to the mean*, which allows for a variable's context to be accounted for in the similarity calculations.

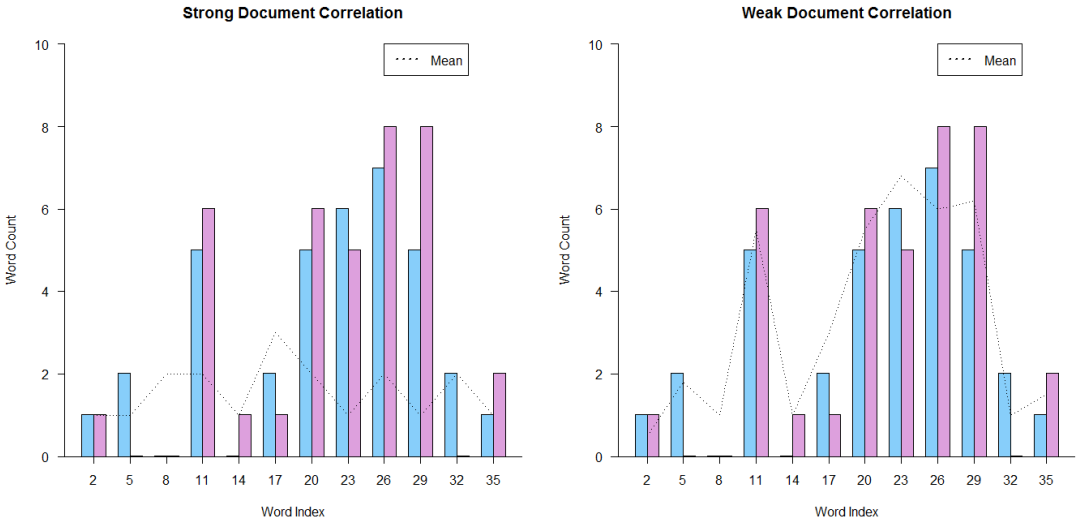


FIGURE 5.3: The left and right figures compare the exact same pair of documents, but with different data means. In the left figure, both documents deviate strongly from the mean in the same variables, yielding strong correlation. In the right figure, the documents generally follow the same pattern as the column means, yielding no significant correlation between the two documents.

This property was highly convenient at first, but we eventually managed to mitigate the impact of such common variables with the IDF column weighting, diminishing the usefulness of data centering within correlation similarity. Also, since most column means are *slightly* above zero in a sparse nonnegative data set, there were many instances of negative similarities (note that Correlation$(x, y) \in [-1, 1]$). We were able to skirt the issue of negativity by re-mapping the similarity matrix from $[-1, 1]$ space to $[0, 1]$ space (subtract the minimum value to get minimum=0, then divide by the new maximum value to get maximum=1). Even then, correlation still failed on one key point: it sacrifices the data sparsity by subtracting the mean. All the zero entries became tiny negative values, since the column means are all slightly above zero. This blew up the memory requirements for calculating a full $n \times n$ similarity matrix. For practical purposes, we absolutely required a similarity measure which would retain the benefits of sparsity.

5.3 Cosine Similarity

In order to preserve sparsity, we required a similarity algorithm which preserved the large proportion of zeros in the dataset. We achieved this by using the Cosine similarity measure:

$$\text{Sim}(x, y) = \frac{\vec{x} \cdot \vec{y}}{\sqrt{||\vec{x}||^2 \, ||\vec{y}||^2}} = \cos(\theta_{xy})$$

Analytically, this is simply the inner product between every pair of rows. Two rows will have a large inner product if they have large values in the same set of variables. If one or both rows have a value close to zero for some variable, that variable will not contribute to their similarity. This has the caveat of strictly finding "positive" similarity, where only simultaneously *large* magnitudes will indicate similarity. Two rows with *low* values in the same set of variables will not demonstrate strong cosine similarity, unless they also have simultaneously large magnitudes in a mutual set of variables as well. This may actually be desirable in a sparse data set, where most rows will probably have zero or near-zero values in a majority of columns.

Geometrically, we are treating each row as a unit vector (dividing by the Euclidean length) and then calculating the angle between each pair of vectors. Similar documents should point in the same "direction", essentially having similar values in a core set of columns (at least, relative to the other documents).

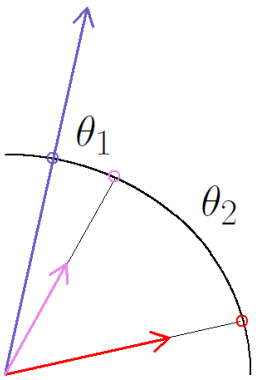


FIGURE 5.4: In this figure, we compare the similarity of the purple vector to the red and blue vectors. Red and purple are closer in terms of Euclidean distance, but the purple and blue vectors are much more similar in terms of their directions (general content). Notice how each vector is projected onto the unit circle (scaled to unit length).

Since our particular dataset is all nonzero values (for both binary and the full count data), each data point will lie in the nonnegative orthant (equivalent to the "first quadrant" in 2D). This gives the extremely convenient consequence of cosine similarities $\in [0, 1]$, useful for Spectral Clustering purposes. Also, to show the computational expedience, we can calculate the cosine similarity extremely simply in matrix notation:

$$\text{Sim}(\boldsymbol{X}) = XX^T$$

where $\boldsymbol{X}$ is the (weighted) data matrix, which should be normalized beforehand to have L2 row lengths equal to 1. Consequently, XX^T is our $n \times n$ similarity matrix, with all the desirable properties of symmetry, nonnegativity, and sparsity. Sparsity can also be further accentuated with certain forms of pre–processing, such as column trimming. The only major drawback of cosine similarity is the potential impact

of abnormally common variables (like common words or popular websites) which could contribute significant but meaningless similarity between rows that are otherwise dissimilar. Two documents which both use the word "the" are not truly similar, but the cosine similarity algorithm will not take into account any variable context (like an abnormally high mean). This highlights the extreme importance of noise reduction in the other steps, mainly column weighting and outlier removal.

Chapter 6

Outlier Removal

A problem that occurs often in clustering task is that there are outlier values that may not cluster well. In order to counter this, we came up with a method to remove these outliers. The idea behind removing outliers was that we could remove documents that were not conducive to clustering, thereby improving the clustering of all other documents.

In our setting, outliers corresponded to documents that:

1. Have low information, or
2. are dissimilar to other documents

Once we determined our outlier criteria, we then had to implement the removal.

6.1 Low Information

In order to remove outliers based on low information, we first had to determine what constituted low information. Because we have been using IDF weighting, we could use that as a measure to determine the value of a document. IDF weighting gives more weight to infrequent words, and less weight to common words, we could sum up all the word values in a document after the weighting step. A document having a low row sum told us one of two things: either the document had very few words, or the words contained in the document were very common. Either of these situations would make a document harder to cluster. ...

6.2 Low Connectivity

As an alternative to removing outliers based purely on their informational value, we removed documents that were not similar to others. In order to do this, we still performed the IDF weighting step on the data, and in addition we then performed cosine similarity. This resulted in us having a symmetric $n \times n$ matrix S where $S_{i,j}$ would be the similarity of document i to document j. When taking the sums

$$d_i = \sum_j S_{i,j}$$

we get the degree of connectivity for each document. If a document has low degree, then it is not similar to many other documents, and therefore is less likely to cluster well.

6.3 Results

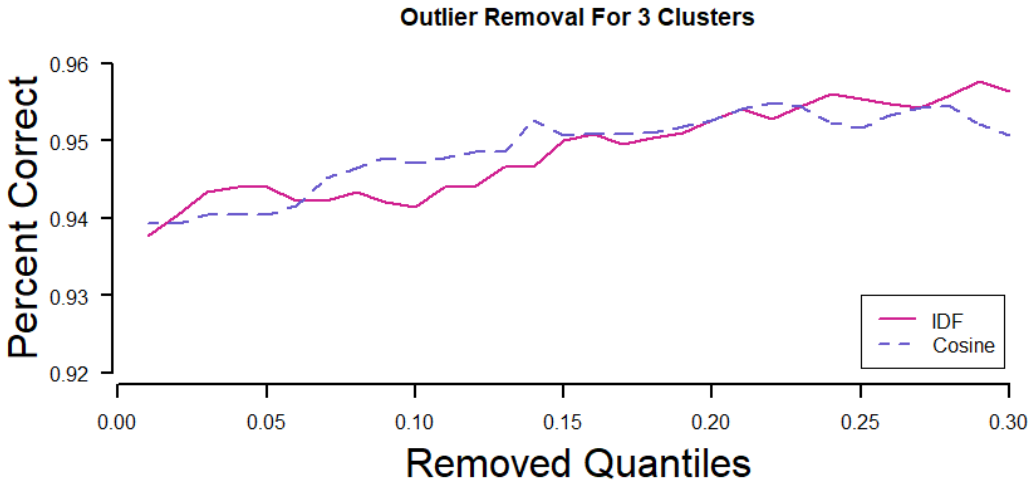


FIGURE 6.1: Outlier Removal for Small Document Set

In figure 6.1, we tested outlier removal for our smallest dataset. This dataset included 3 clusters that were fairly distinguishable from each other, allowing us to have nice computational efficiency. One thing we noticed for this dataset was that removing documents with low connectivity and documents with low information gave us similar results.

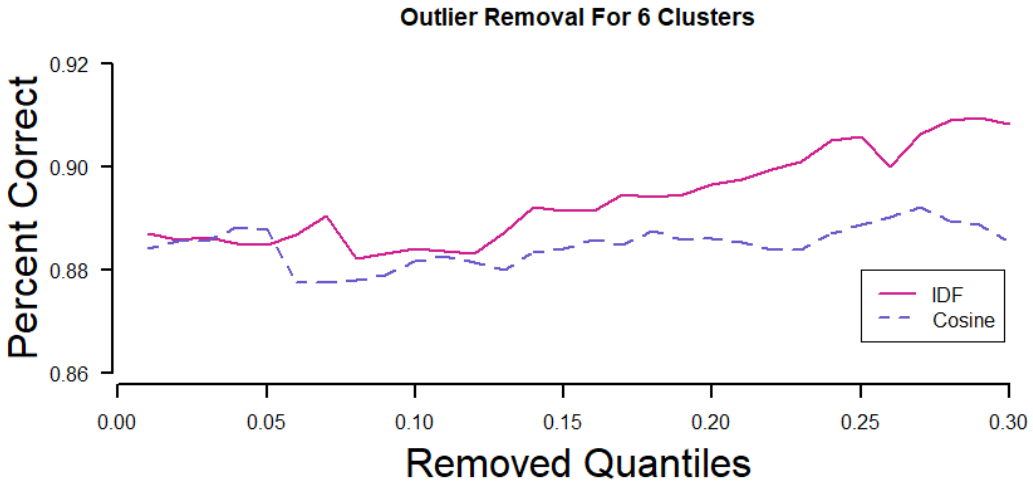


FIGURE 6.2: Outlier Removal for Large Document Set

In figure 6.2 we tested outlier removal for 6 known clusters, one from each meta-cluster. One thing to note is that as we removed larger percentages of the data, we got consistently better results removing documents with low information than we did removing documents with low connectivity.

Chapter 7

Spectral Clustering

Now that we have all our preprocessing done and our similarity matrix we move on to clustering our observations. The clustering method we spent most of our time on and had the greatest success with was spectral clustering. There are three different spectral clustering algorithms that we used in our project; normalized cut(NCut), Ng, Jordan, Weiss(NJW), and diffusion maps. These three algorithms are very similar with slight variations which we will go over in the next sections.

7.1 Normalized Cut

The first spectral clustering method we tried was the normalized cut(NCut) and is probably the mostly commonly used. One common way to think about spectral clustering is as a graph cut problem or a method of finding an optimal way of removing edges from a graph so we separate our observations into different groups. If you think of the similarity matrix we constructed earlier as a graph showing the connectivity from one point to another what we want to do is remove these connections in some optimal way.

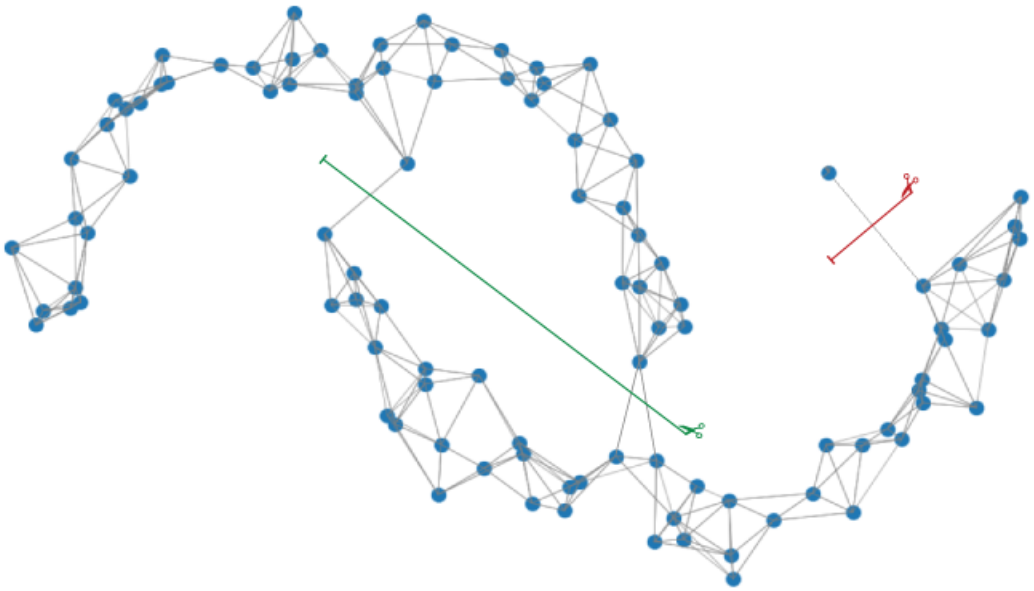


FIGURE 7.1: An example of a graph constructed from a similarity matrix

For example, in Figure 7.1 if we simple removed the fewest number of edges or gray lines till we had two separate clusters we would wind up with the cut represented by the red scissor. This cut is not ideal since we end up with one point by itself in one cluster and everything else in another cluster. A better way the separate the points is to use the green cut which balanced the number of edges we are removing with the resulting cluster size. This is the method suggested by Shi and Malik[13]. The formula for this is

$$\min\left(\frac{Cut(A,B)}{Vol(A)}+\frac{Cut(A,B)}{Vol(B)}\right)$$

Where Cut(A,B) is the sum of the edges we are removing and Vol(B) is the sum of all the edges in cluster B or A.

Solving for this would be extremely difficult since we would have to check every possible way of removing points to minimize this. We will be using linear algebra to find an approximate solution but first lets define some terms

Let

$$x_i=\begin{cases}1\text{, if the observation i is in cluster A}\\ -1\text{, otherwise}\end{cases}\qquad\text{for all observations i.}$$

and

$$\mathbf{D}=\begin{bmatrix}d_1 & & 0\\ & \ddots & \\ 0 & & d_n\end{bmatrix}\qquad\text{where } d_i=\sum_{j=1}^{n}w_{ij}$$

If we also let

$$\mathbf{y}=(1+\mathbf{x})-\frac{\sum_{\mathbf{x}_i>0}d_i}{\sum_{\mathbf{x}_i<0}d_i}(1-\mathbf{x})\qquad\text{and}\qquad\mathbf{L}=\mathbf{D}-\mathbf{W}$$

It can be shown that

$$\min\left(\frac{Cut(A,B)}{Vol(A)}+\frac{Cut(A,B)}{Vol(B)}\right)=\min_{\mathbf{y}}\frac{\mathbf{y}^T\mathbf{L}\mathbf{y}}{\mathbf{y}^T\mathbf{D}\mathbf{y}}$$

If we relax the requirement that the entries of **x** need to be 1 or -1 and instead require the entries to be real numbers this becomes the minimization of a quadratic form. Thus,the solution to the normalized cut problem can be approximated by the sign of the second largest eigenvector of $\mathbf{D}^{-1}\mathbf{L}$

In general to find more than two clusters we only need to take more eigenvectors of $\mathbf{D}^{-1}\mathbf{L}$. Then we only need to run a simple clustering method such as kmeans to find the clusters.

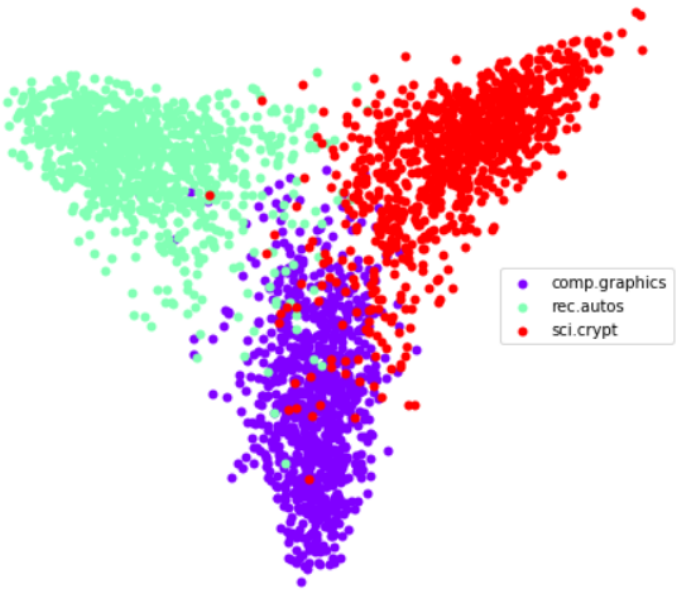


FIGURE 7.2: The $\mathbf{V}$ matrix for the 3 clusters subset of 20 news group dataset using Ncut

Algorithm (Ncut)

1. Construct similarity matrix $\mathbf{W}$
2. $\mathbf{L} = \mathbf{D} - \mathbf{W}$
3. Find the first k eigenvectors of $\mathbf{D}^{-1}\mathbf{L}$
4. Make a matrix $\mathbf{V}$ by stacking the 2nd to kth eigenvectors
5. Cluster using kmeans using $\mathbf{V}$ where each row represents a point

7.2 Ng, Jordan, Weiss

Another method suggested by Ng, Jordan and Weiss [9] is very similar to the Ncut algorithm described above with only a few small changes. The first change is instead of finding the eigenvectors of $\mathbf{D}^{-1}\mathbf{L}$ we find the eigenvectors of $\mathbf{D}^{-1/2}\mathbf{L}\mathbf{D}^{-1/2}$. The second is after finding the matrix $\mathbf{V}$ we then normalize the rows of $\mathbf{V}$

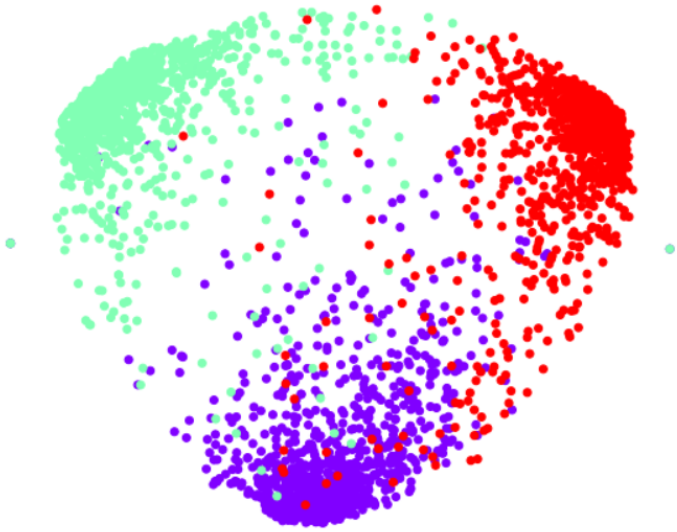


FIGURE 7.3: The **V** matrix for the 3 clusters subset of 20 news group dataset using NJW

Algorithm (NJW)

1. Construct similarity matrix **W**
2. $\mathbf{L} = \mathbf{D} - \mathbf{W}$
3. Find the first k eigenvectors of $\mathbf{D}^{-1/2}\mathbf{L}\mathbf{D}^{-1/2}$
4. Make a matrix **V** by stacking the k eigenvectors
5. Normalize the rows of **V**
6. Cluster using kmeans using **V** where each row represents a point

7.3 Diffusion Maps

The idea of diffusion maps is to use eigenvectors of Markov matrices to construct coordinates called diffusion maps that generate efficient representations of complex geometric structures [2].

It is connected with spectral clustering through the random walk explanation of the latter.

A transition matrix $P = (p_{ij})_{i,j=1,\ldots,n}$ of the random walk is defined by [16]

$$P = D^{-1}W \tag{7.1}$$

As described by Von [16], p_{ij} represents the probability of transition in one step from point i to point j, and it's proportional to the edge weight w_{ij}. Spectral Clustering Ncut method is equivalent to the transition probabilities of random walk.

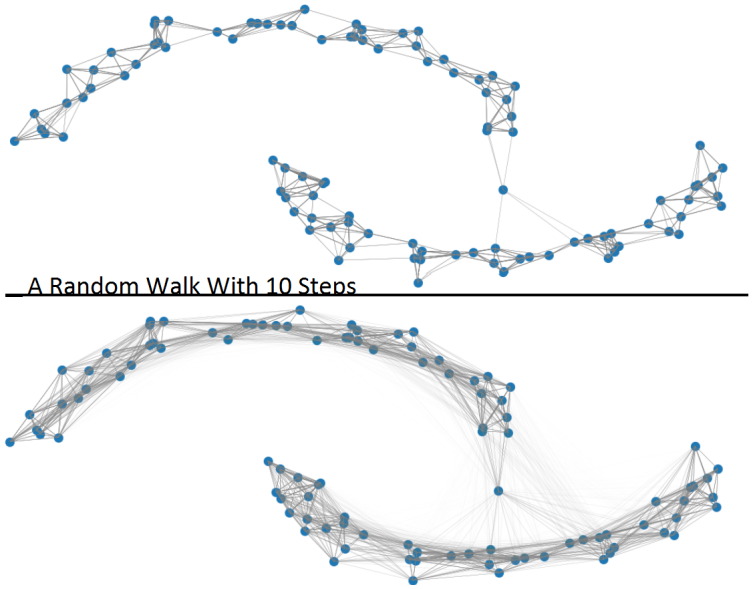


FIGURE 7.4: Diffusion Maps

Diffusion maps is to take the different power of transition matrix and reveal the probability of moving from one point to another point in t time steps. It allows to integrate the local geometry and reveal relevant geometry structures of data in different scales.Figure 7.4 shows the regular random walk and random walk with 10 steps. With more steps, the points that far from each other can be connected.

The power of P matrix have different eigenvalues, and t changes the number of significant eigenvalues. In Figure 7.5, for the same cutoff 0.1, as t increase, the number of significant eigenvalues decrease.

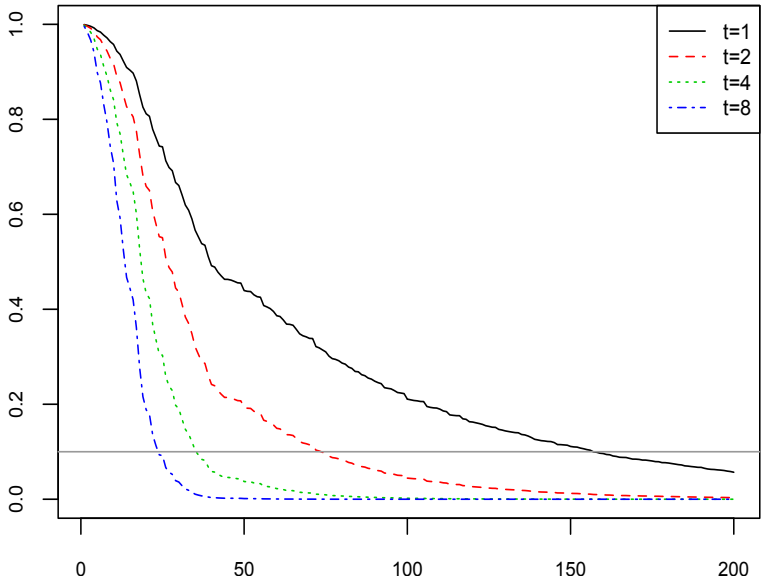


FIGURE 7.5: Eigenvalues of P^t

Let $\{\lambda_l\}_{l\geq 0}$ denotes the eigenvalues of P, and $\{v_l\}_{l\geq 0}$ represents the eigenvectors of P. The diffusion maps $\{V_t\}_{t\in N}$ is given by

$$V_t(x) = \begin{pmatrix} \lambda_1^t v_1(x) \\ \lambda_2^t v_2(x) \\ \dots \\ \lambda_k^t v_k(x) \end{pmatrix}$$

The new coordinates is now in the Euclidean space, so we can calculate the new space L2 distance, and feed it into the Kmeans clustering algorithm. Figure 7.6 is a diffusion maps clustering result on the 3 clusters subset of 20 news group data. The larger t tends to make the clusters fuse together.

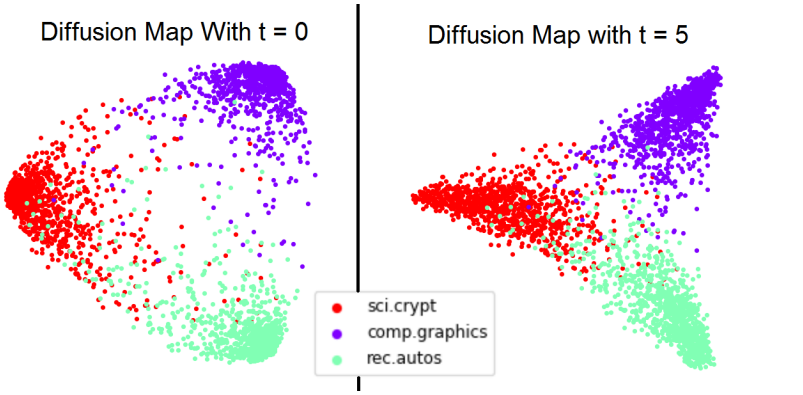


FIGURE 7.6: 3 clusters subset(sci.crypt, comp.graphics, rec.autos) of 20 news group dataset by Diffusion Maps

Here is the algorithm of diffusion maps clustering, which is a modification on the Ncut algorithm by changing the calculation of eigenvectors.

Algorithm (Diffusion Maps)

1. Construct similarity matrix **W**
2. **L** = **D** − **W**
3. Find the first k eigenvectors of $\mathbf{D}^{-1}\mathbf{L}$
4. Make a matrix **V** by stacking the 2^{nd} to k^{th} eigenvectors
5. $V = (\lambda_1^t v_1, \lambda_2^t v_2, ..., \lambda_k^t v_k)$
6. Normalize the rows of **V**
7. Cluster using Kmeans using normalized **V** where each row represents a point

Chapter 8

Insights

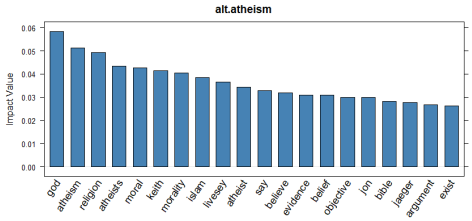
Since we can now cluster our data into groups with common features, it could be useful for us to have some insights into what features are common to each cluster. In order to do this, we will use Principal Component Analysis to determine which features are most prominent for each cluster.

8.1 Results from 20 Newsgroups Data

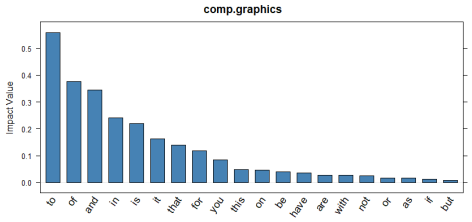
For the 20 Newsgroups data, we had the truth of our clusters known. With that truth, in order to test the concepts, we used SVD with $k = 1$ to extract the most important vector of words for each cluster. When looking at the SVD factorization

$$XX^T = USV^T$$

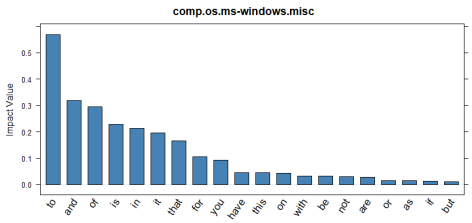
we remember that V provides an orthonormal basis for the column space associated with our cluster. With $k = 1$ we are taking the vector v most associated with the cluster we are observing.



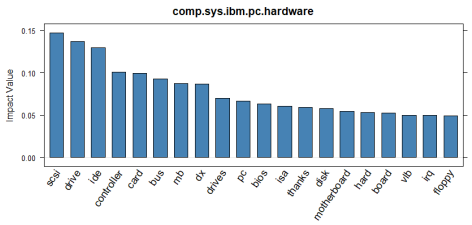
(A) alt.atheism



(B) comp.graphics

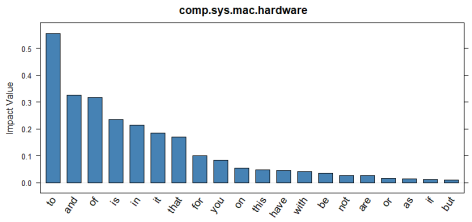


(C) comp.os.windows.misc

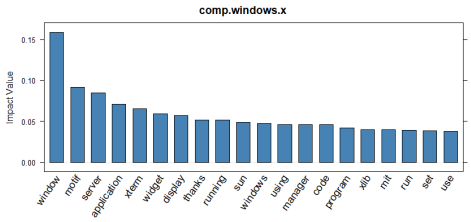


(D) comp.sys.ibm.pc.hardware

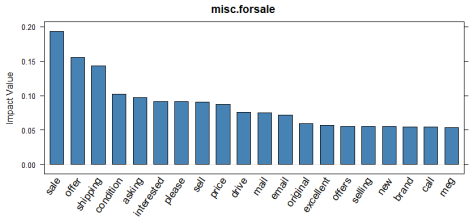
FIGURE 8.1: Top 20 keywords for newsgroups 1-4



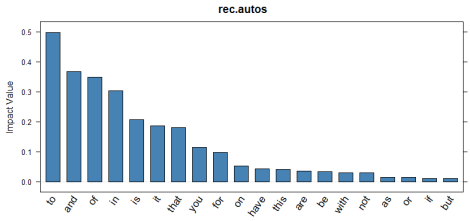
(A) comp.sys.mac.hardware



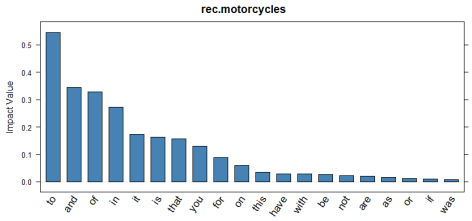
(B) comp.windows.x



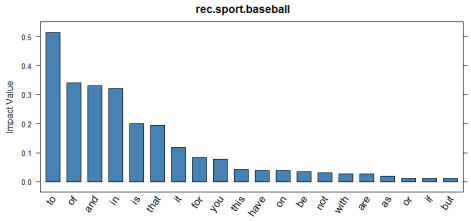
(C) misc.forsale



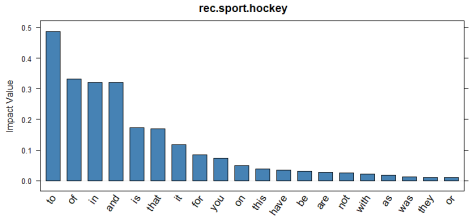
(D) rec.autos



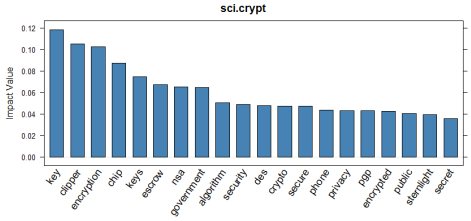
(E) rec.motorcycles



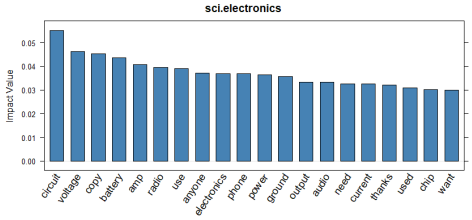
(F) rec.sport.baseball



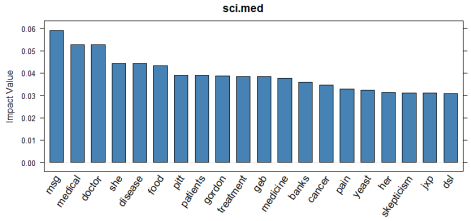
(G) rec.sport.hockey



(H) sci.crypt

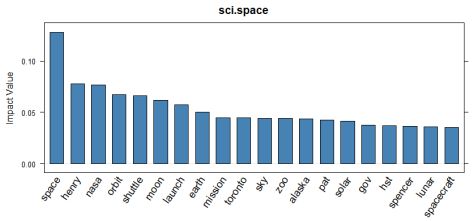


(I) sci.electronics

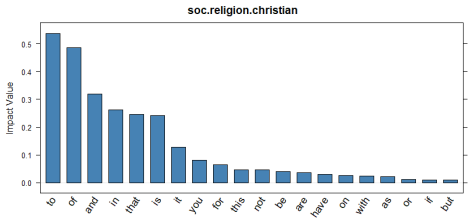


(J) sci.med

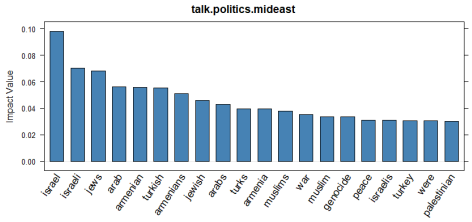
FIGURE 8.2: Top 20 keywords for newsgroups 5-14



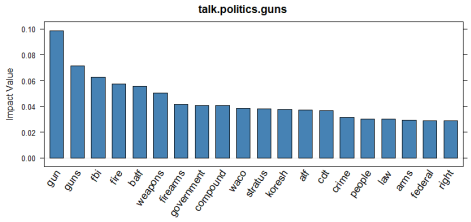
(A) sci.space



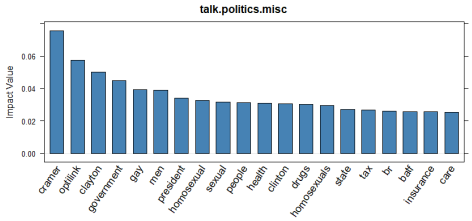
(B) soc.religion.christian



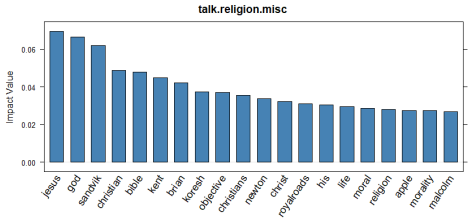
(C) talk.politics.mideast



(D) talk.politics.guns



(E) talk.politics.misc



(F) talk.religion.misc

FIGURE 8.3: Top 20 keywords for newsgroups 15-20

Figures 8.1-8.3 shows the top 20 words into each of our 20 newsgroups. One thing to notice is that while some of the newsgroups seem very accurate, knowing what they are, others have a lot of meaningless words (to, or, and, it, etc.). These *stopwords* can obscure the true insights into some of our clusters. Notably obscured insights occur in *comp.windows.x*, in Figure 8.2B, as well as in the rec newsgroups.

8.2 Potential for Application to Verizon Data

Because one of the goals for the Verizon data set is market segmentation, we thought that this method of gathering insights could be applied after clustering with the intent of utilizing human insight into the various clusters. For example: if a cluster were formed, and the principal "direction" of this cluster led you to websites for tractors, feed, and Cabela's, a human would realize that we're dealing with a cluster of farmers.

Chapter 9

Results

9.1 20 Newsgroup Results

9.1.1 Measurement

There are plenty of measurements to define the algorithms work well or not, such as Accuracy, Adjusted Rand Index (ARI) [10] [15], or F-measure. The Adjusted Rand Index, which is adjusted from the Rand index, is to compare data clusterings by using contingency tables 9.1. $X_1, X_2, \ldots, X_r$ and $Y_1, Y_2, \ldots, Y_s$ are represented two clusterings of these points. Also, n_{ij} means the intersection between X and Y.

TABLE 9.1: The contingency table.

XY	Y_1	Y_2	...	Y_s	Sums
X_1	n_{11}	n_{11}	...	n_{1s}	a_1
X_2	n_{21}	n_{22}	...	n_{2s}	a_2
...	...	...	...	...	...
X_r	n_{r1}	...	...	n_{rs}	a_r
Sums	b_1	b_2	...	b_s	

The below is the formula of ARI. The values of a_{ij}, b_{ij}, and n_{ij} are from the contingency table which introduced before.

$$ARI = \frac{\sum_{ij}\binom{n_{ij}}{2} - [\sum_j \binom{a_i}{2} \sum_j \binom{b_j}{2}] \div \binom{n}{2}}{\frac{1}{2}[\sum_i \binom{a_i}{2} + \sum_j \binom{b_j}{2}] - [\sum_j \binom{a_i}{2} \sum_j \binom{b_j}{2}] \div \binom{n}{2}}$$

Since we have the ground truth for 20 Newsgroup dataset, we focus on the accuracy to test our proof of concept implementation. The accuracy is th percentage of data points that are truly in the same cluster are predicted to be in the same cluster.

On the other hand, we also record how much time our implementation takes to evaluate the efficiency. The running time is recorded from the beginning, data processing, to the end, spectral clustering, including getting the insights. For the first tasks, the results can done on a laptop equipped with a 2.4 GHz Intel i5 (Skylake) dual core processor. For the remaining two tasks, the results can be done on a workstation equipped with two 2.4 GHz Intel Xeon (Westmere) quad core processor.

9.1.2 Performance

Table 9.2 is the summary of the results. These results are applying cosine similarity without SVD. In general, IDF^2 and Ncut or Diffusion Map performs better than other combinations, except the fourth task. Our algorithm performs well in the most of cases.

TABLE 9.2: The accuracy of 6 Tasks without SVD.

ColWeight	Clustering	1st	2nd	3rd	4th	5th	6th
IDF	NJW	93.83%	63.40%	41.03%	88.74%	49.91%	55.16%
IDF	Ncut	94.00%	69.45%	39.37%	88.27%	45.55%	54.54%
IDF	Diffusion Map	94.00%	66.43%	41.06%	88.24%	49.17%	54.27%
IDF^2	NJW	95.42%	69.24%	39.39%	78.52%	56.34%	59.01%
IDF^2	Ncut	85.80%	61.18%	40.56%	76.95%	49.86%	60.26%
IDF^2	Diffusion Map	73.12%	69.79%	37.04%	72.31%	56.43%	60.91%

After plotting the results, we could expect that for the easier tasks, such as the first and the fourth task, the algorithm could attain 88% to 95% accuracy. For the harder task, such as the second task and the third task, the algorithm could get around 41% to 70% accuracy. For the full data, the algorithm could achieve 56% to 61% accuracy.

FIGURE 9.1: The accuracy of 6 Tasks without SVD

Table 9.3 is the other table of the summary of the results. These results are applying cosine similarity with SVD. In general, NJW performs better than other combinations when applying IDF column weighting. Nut and Diffusion Map perform better when applying IDF^2 column weighting.

TABLE 9.3: The accuracy of 6 Tasks without SVD.

ColWeigh	Clustering	1st	2nd	3rd	4th	5th	6th
IDF	NJW	94.62%	89.92%	47.01%	87.49%	51.55%	60.90%
IDF	Ncut	95.02%	67.44%	44.64%	87.08%	46.49%	60.99%
IDF	Diffusion Map	94.45%	68.03%	44.54%	86.67%	46.01%	61.02%
IDF^2	NJW	95.25%	90.34%	62.32%	86.11%	55.61%	62.00%
IDF^2	Ncut	95.59%	89.29%	53.30%	86.48%	61.58%	70.70%
IDF^2	Diffusion Map	95.53%	90.04%	53.22%	86.42%	60.90%	62.27%

After including SVD step on our data, we could see the trends of the accuracy lines are similar to the previous results without SVD in the following Figure 9.2. The results with SVD are better than those without SVD, with improvement ranging from absolute 1% to 21%. For the harder tasks, such as the second, the third task and the full data, the accuracies are increasing significantly.

FIGURE 9.2: The accuracy of 6 Tasks with SVD

Our algorithm is also efficient. After plotting the running time, we could expect that for the small tasks, such as the first four tasks, the algorithm could spend less than 1 minutes to obtain the results. For the full data, such as the fifth task and the sixth task, the algorithm could spend less than 5 minutes to get the results.

In general, although IDF^2 and Ncut or Diffusion Map spend more time than other other combinations, it only spend less than 5 minutes in total. There is not much differences among clustering methods in Figure 9.3.

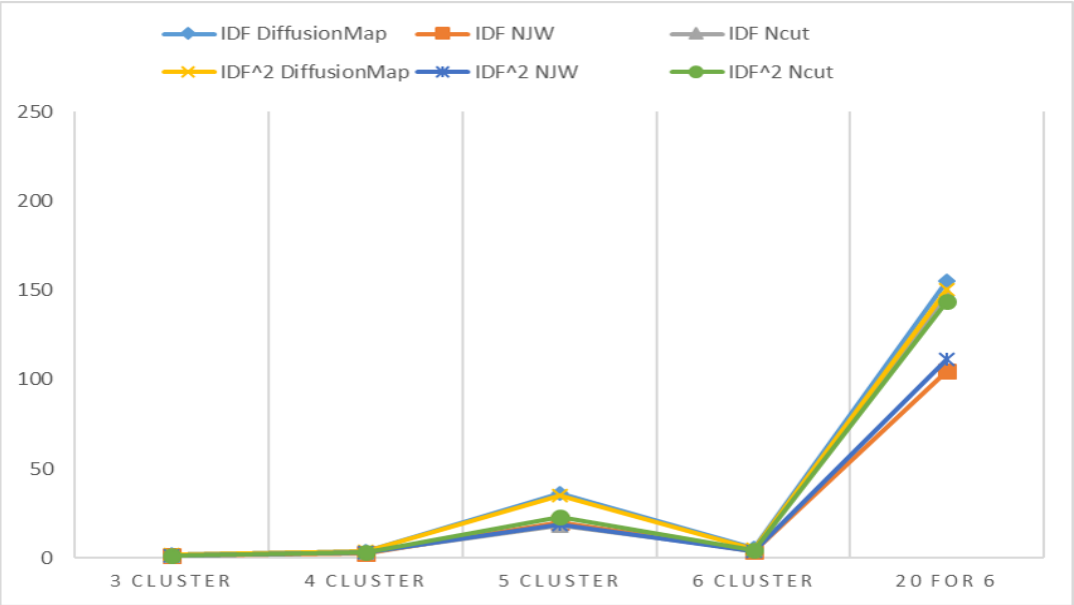


FIGURE 9.3: The running time of 6 Tasks

9.2 Study Sensitivity of Parameters

Since there are different setting for our algorithm, such as dimensions in SVD and steps in Diffusion Map, we would provide the results to discuss the performance and decide which parameters we choose for applying Verizon data later.

9.2.1 Different dimensions in SVD

From Figure 9.4, with IDF^2 as column weighting method and diffusion Map as clustering method, we find that for the four tasks, the accuracies in different dimensions are not differ significantly. For the first task, the accuracy are around 93% to 96%. For the second task, the accuracy are around 68% to 91%. For the third task, the accuracy are around 57% to 64%. For the forth task, the accuracy are around 84% to 89%.In most cases, we can see SVD with 200 or 250 dimensions performs better than other dimensions.

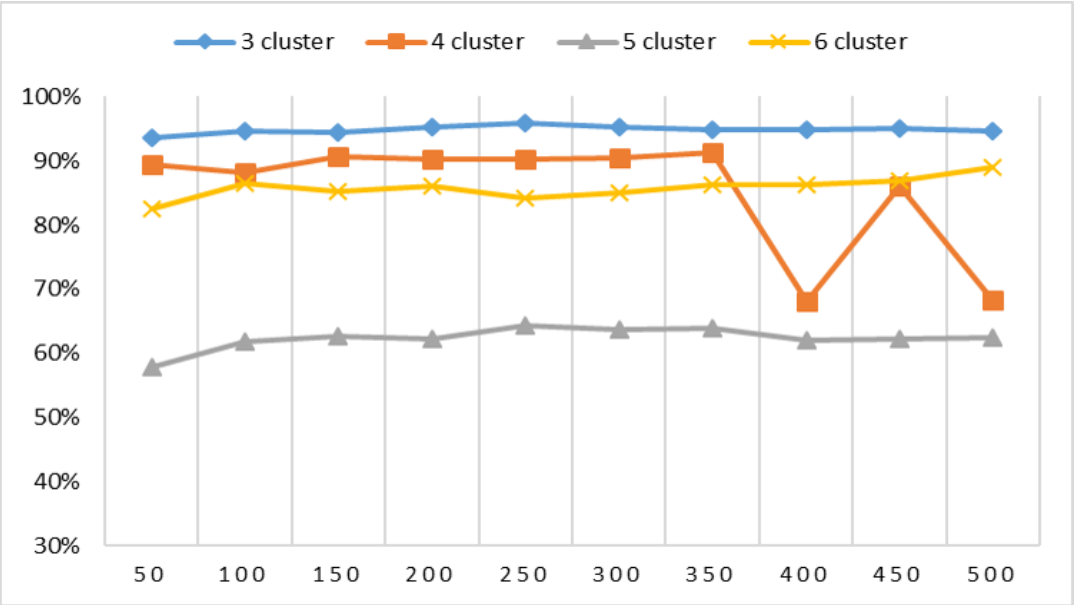


FIGURE 9.4: Different dimensions in SVD with IDF^2 column weighting method.

In general, we can see that after 300 dimensions, the accuracy would be decreasing. Considering to the performances, we think that for sparse data, such as 20 newsgroup data or Verizon data, we probably use 200 dimensions instead of more dimensions for getting better results.

9.2.2 Different steps in Diffusion Map

From Figure 9.5, with IDF as column weighting method, we find that for the first task, the accuracy are around 91% to 93%. For the second task, the accuracy are around 63% to 67%. For the third task, the accuracy are around 39% to 41%. For the forth task, the accuracy are around 41% to 88%. For the fifth task, the accuracy are around 36% to 49%. For the sixth task, the accuracy are around 16% to 51%.

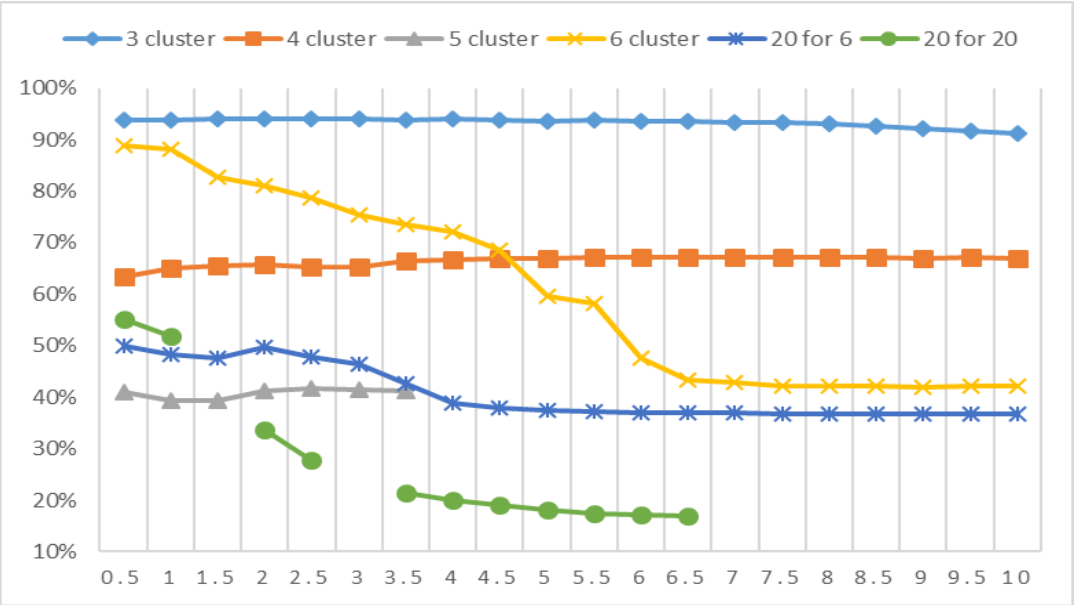


FIGURE 9.5: Different T in Diffusion Map with IDF column weighting method.

In general, we can see that after one or two steps, which means t equals to 0.5 or 1, the accuracy would be decreasing. Moreover, for the third task and the sixth task, the performances are too bad for our algorithm to calculate the accuracy after few steps. Considering to the performances, we think that for sparse data, such as 20 newsgroup data or Verizon data, we probably use one step or two step instead of more steps for getting better results.

9.3 Verizon Results

9.3.1 Determine number of clusters

Since we do not have ground of truth for Verizon data, we would use internal evaluation to decide numbers of clusters. The ideal clustering method usually produces clusters with high similarity within a cluster and low similarity between clusters. We apply basic k-means to explore numbers of clusters. In Figure 9.6, there is no obvious elbow turning point.

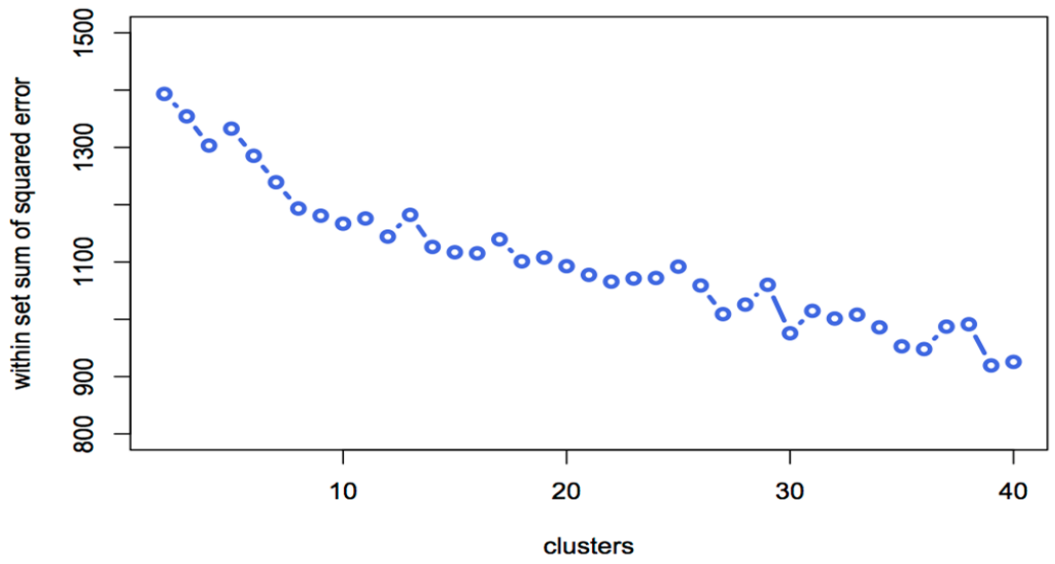


FIGURE 9.6: Determine numbers of clusters for Verizon data.

We also use another method, which is the ratio of within-clusters variation over between-cluster variation, to detect number of clusters. [7] We simply try different number of clusters and compare the results. We can identify 18 clusters for Verizon data from Table 9.4.

TABLE 9.4: The SSWithin and SSBetween.

Number of cluster	Within-cluster Variation	Between-cluster Variation
3	10,253	29,678
12	15,816	44,807
18	10,283	36,911

9.3.2 Full data SVD visualization

We use the first three dimensions of SVD to visualize the full Verizon data. In Figure 9.7 we find that there seems some clusters for Verizon data. After rescaling the three dimensions, we can clearly see there are at least three clusters for Verizon data in Figure 9.8. For Verizon data, SVD still plays a good role to visualize the data points.

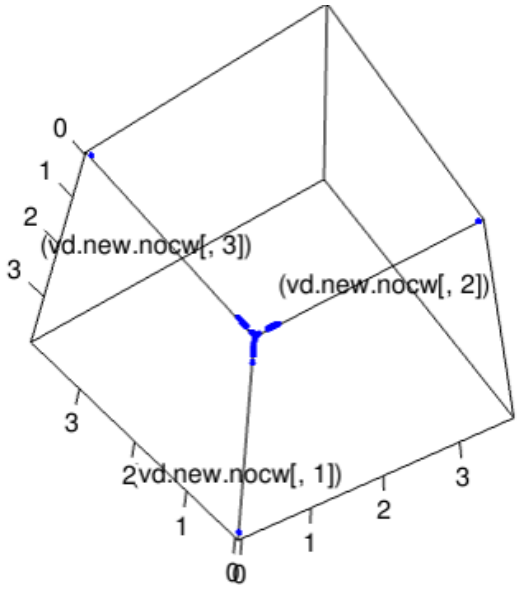


FIGURE 9.7: Original Scale for full data.

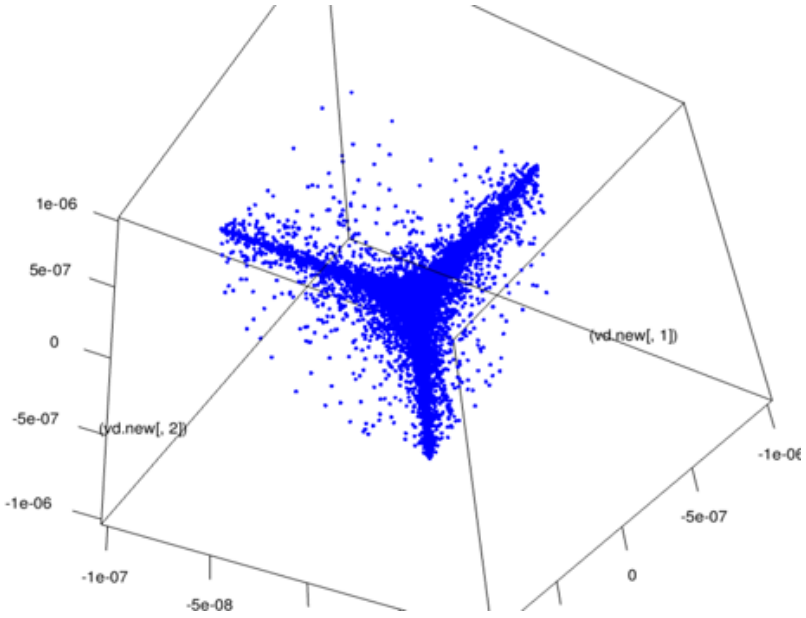


FIGURE 9.8: Rescale by 10^{-3} for full data.

9.3.3 Some trial Results

From Figure 9.8, we can find three clusters. We apply our algorithm, which use cosine similarity, IDF as column weighting, and NJW as clustering method. From Figure 9.9, we can see our algorithm can separate the data points into three clusters. We can expect that our algorithm is useful for Verizon data.

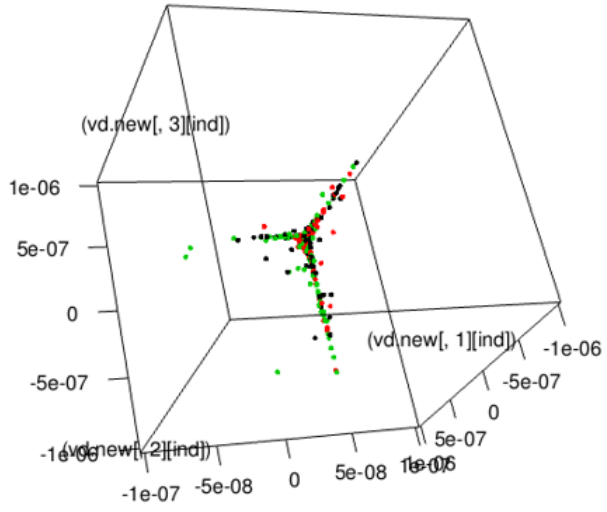


FIGURE 9.9: The Visualization for the clustering result.

Besides, we focus on specific variable, dmaMAP, to check the algorithm work well or not like what we did for the subsets of 20 newsgroup data. To see the clusters clearly, we rescale the data points by 10^{-6}. Using the same visualization method, SVD with 10 dimensions, we could see there seems four clusters for dma501 data in Figure 9.10.

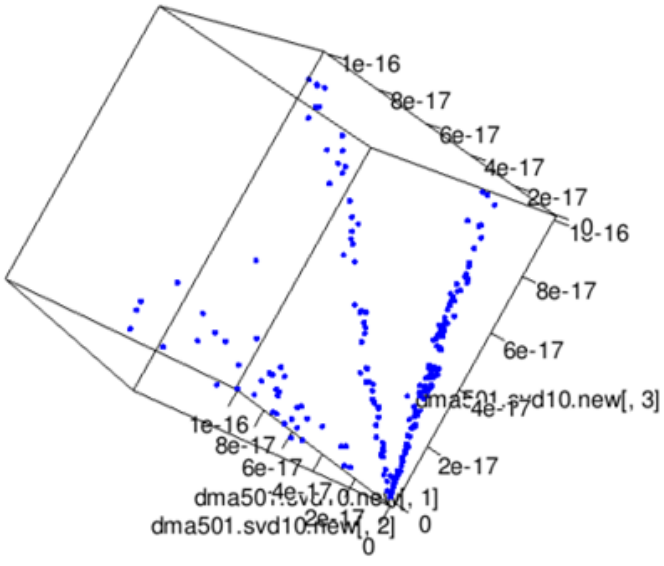


FIGURE 9.10: The Visualization for the data of dma501.

After applying our algorithm for the subset data, which use cosine similarity, no column weighting, NJW as clustering method, and SVD, we could attain the following Figure 9.11. We find that our algorithm work well on this subset. The majority of the data could be separated clearly.

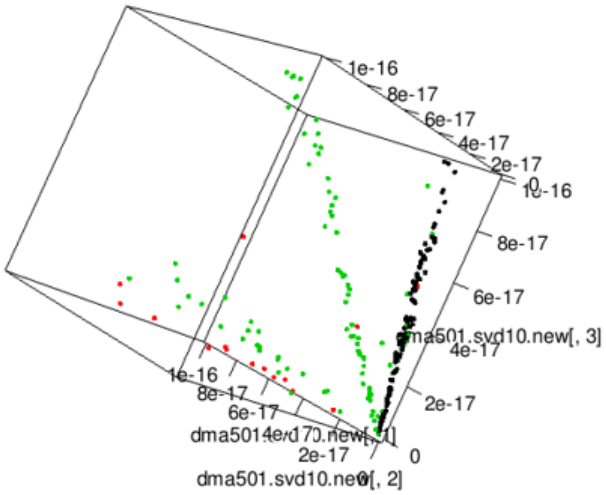


FIGURE 9.11: The Visualization for the clustering results for the data of dma501.

Chapter 10

Future Work

There were many ideas that we left unexplored, due to limited time and attention. Many of them seem promising in theory, addressing some of the shortcomings we acknowledge in our own method.

10.1 SVD Approximation of Cosine Similarity

The slowest part of the algorithm is constructing the similarity matrix, generally requiring the matrix multiplication or manipulation between two extremely large matrices. Cosine similarity is our fastest similarity algorithm, where the only intensive operation is the simple matrix multiplication XX^T. Many languages are optimized for this sort of computation, yet CPU limitations will inherently hinder efficiency when X could feasibly contain millions of rows and columns in practical situations. Fortunately, XX^T is not the object of interest. Rather, it is one step toward the eventual eigendecomposition of $D^{-1/2}WD^{-1/2}$ in the Diffusion Map algorithm. Recall that W is the similarity matrix but with zeroes on the diagonal ($W = XX^T - I_n$, where X has been L2 row-normalized). D is the "Degree matrix", a diagonal matrix with elements equal to the rowsums of the similarity matrix W. Algebraically, we can manipulate this object:

$$\begin{aligned} D^{-1/2}WD^{-1/2} &= D^{-1/2}\Big(XX^T - I_n\Big)D^{-1/2} \\ &= D^{-1/2}\Big(XX^T\Big)D^{-1/2} - D^{-1/2}\Big(I_n\Big)D^{-1/2} \\ &= \Big(D^{-1/2}X\Big)\Big(D^{-1/2}X\Big)^{\mathrm{T}} - D^{-1} \end{aligned}$$

since $D^{-1/2}$ is a diagonal matrix, making the transpose trivially equivalent. Therefore,

$$\begin{aligned} \text{eigen}\Big(D^{-1/2}WD^{-1/2}\Big) &= \text{eigen}\Bigg(\Big(D^{-1/2}X\Big)\Big(D^{-1/2}X\Big)^{\mathrm{T}} - D^{-1}\Bigg) \\ &= \text{eigen}\Bigg(\Big(D^{-1/2}X\Big)\Big(D^{-1/2}X\Big)^{\mathrm{T}} - d^{-1}\cdot I_n\Bigg) \\ &= \text{eigen}\Bigg(\Big(D^{-1/2}X\Big)\Big(D^{-1/2}X\Big)^{\mathrm{T}}\Bigg) - d^{-1} \\ &= \text{SVD}\Big(D^{-1/2}X\Big) - d^{-1} \end{aligned}$$

if and only if $D^{-1} = d^{-1}\cdot I_n$, a scalar multiple of the identity matrix. This tells us that the eigenvectors of $D^{-1/2}WD^{-1/2}$ can be found exactly by the Singular Value

Decomposition of $D^{-1/2}X$, and the eigenvalues are the same except subtracted by the scalar d^{-1}. In practice, d rarely exists – the rowsums of XX^T aren't generally constant. However, this approximation should be accurate up to the extent that D approximates a constant matrix. To check this condition, it is actually possible to calculate the degree matrix D and inspect it, without ever directly computing XX^T:

$$\begin{aligned}\text{rowsums}(XX^T - I) &= \left(XX^T - I_n\right) \cdot \vec{1} \\ &= XX^T \cdot \vec{1} - I_n \cdot \vec{1} \\ &= X\left(X^T \cdot \vec{1}\right) - \vec{1}\end{aligned}$$

This clever usage of the associative property avoids ever computing the giant matrix multiplication of XX^T. Instead, we compute the very simple matrix–vector multiplication $X^T \cdot \vec{1}$, which yields a $n \times 1$ vector (call it $\vec{Y}$), and then we do a second matrix–vector multiplication $X\vec{Y}$. Subtracting 1 from the diagonal of $X\vec{Y}$ produces the degree *vector* of rowsums, which contains all of the relevant information.

It should also be mentioned that as d becomes large (e.g. when X has many rows, then XX^T gathers more nonzero columns), the scalar value d^{-1} tends to zero and the subtraction of d^{-1} from the eigenvalues becomes insignificant. In essence, this approximation works great if we have gathered enough observations (with sufficient similarity between them), or if all the rows have equal total connectivity. Certain forms of outlier treatment can also be implemented to help achieve uniform connectivity between rows, removing any rows which have abnormally small or large connectivity. If we can validate one of these two assumptions, then clever usage of SVD could thereby avoid the direct computation of XX^T altogether, bypassing the slowest part of our algorithm.

10.2 Landmark Centers for Similarity

In practice, the SVD approximation will almost always carry some error, especially considering that most "real life" cases will have high variability across D, the rowsums of the similarity matrix W. A different approach toward expediting the XX^T process involves downsizing the X^T matrix, to lighten the matrix multiplication. XX^T calculates the inner product between every pair of rows in X, such that the $(i,j)^{th}$ entry of XX^T represents the inner product of row X_i with row X_j. We then use the XX^T matrix to look for groups of rows which are all mutually similar to each other and dissimilar everywhere else.

FIGURE 10.1: Here we see a heatmap of the $n \times n$ cosine similarity matrix for three subclusters of the 20News dataset. As expected, we can see a pattern of three distinct internally-connected squares in the bottom–left, middle, and bottom–right areas. These disjoint groups form the basic clustering structure. The signal is not of ideal strength, but even this faint signal produced 95% accuracy when used with Diffusion Maps spectral clustering.

This result works perfectly well, but we realized that this $n \times n$ similarity matrix contained far more information than necessary. In reality, we could retrieve the same clustering information with only a tiny subset of these columns.

Sample Similarity Matrix

1	1	1	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0
0	0	0	1	1	1	0	0	0
0	0	0	1	1	1	0	0	0
0	0	0	1	1	1	0	0	0
0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	1	1	1

Randomly Selected "Centers"

1	1	1	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0
0	0	0	1	1	1	0	0	0
0	0	0	1	1	1	0	0	0
0	0	0	1	1	1	0	0	0
0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	1	1	1

Subset of Similarity Matrix

1	0	0
1	0	0
1	0	0
0	1	0
0	1	0
0	1	0
0	0	1
0	0	1
0	0	1

FIGURE 10.2: The left figure is a sample similarity matrix in the "ideal" case, where similarities equal 1 between observations of the same cluster and equal 0 otherwise. The middle figure highlights a sample of three columns to be selected from the 9 total columns. The right figure is the resultant three–column subset. Notice how the left table and the right table contain essentially the same information, despite the right table having only $\frac{1}{3}$ as many rows.

In the figure above, we see that not all n-many columns are needed to extract the similarity information. The right table is just the result of taking the inner product of

all n rows with only the 1^{st}, 6^{th}, and 8^{th} rows, as seen in the middle table where those three rows (which are columns in XX^T) are highlighted. Put more simply, we only need a small subset of "landmark" rows to compare against; the other information becomes redundant once we have a good subset of these landmark centers.

Once the subset matrix is taken – let's call it Y – you can construct the matrix YY^T, which compares each pair of rows in terms of their similarity to the chosen landmark centers, hopefully recovering the full cluster membership information. This YY^T matrix is square and symmetric, allowing for easy eigenvalue calculation and compatibility with the various spectral clustering algorithms. In general, this may allow for implementation of methods that are not inherently speedy, such as distance–based or non–linear methods.

We did not dedicate time toward find a method for intelligently identifying landmark centers, as our full algorithm already ran very efficiently on our training datasets. However, we did find that taking a random sample of rows was usually quite effective, as long as you sampled around 10–20% of rows. The amount of rows necessary for a strong random sample is proportional to the strength of connectivity/similarity in the dataset. In the dataset referenced by Figure 10.1, we recovered almost the full accuracy with only 10% of rows sampled. We found that this method required 20% of rows to be sampled in a 6–cluster subset or in a inter–newsgroup dataset (such as the ".rec" or ".comp" newsgroup subsets) to recover near–complete accuracy. 20% is the recommended *random* sample size for this approach in a cautious implementation, although smaller sample sizes may be sufficient if efficiency must be minimized.

Another option is to iterate the process. We could first run the algorithm with these random centers, finding some preliminary cluster partition. Once we have a "first guess" at cluster memberships, we can use this to inform a second implementation of random centers similarity, taking a new subset of rows from each cluster as a new set of landmark centers. This should allow fair representation of each cluster, with representation roughly proportional to the size of the cluster. This stratified sampling may smooth out any potential misrepresentation that is possible with a completely random, non–stratified sample. It may also allow for smaller subsets to be sampled; we speculate that multiple iterations could allow very small sampling proportions, with perhaps only 5% or smaller sampling required.

10.3 Feature Clustering

One popular approach for high–dimensional datasets is to try grouping common columns together. There is often a lot of redundant information contained in high–dimensional clustering datasets. For example, the 20 Newsgroup dataset has distinct columns for the words "car", "automobile", and "vehicle". These words are almost certainly correlated with each other, and their multiple presences don't add any new information to the data.

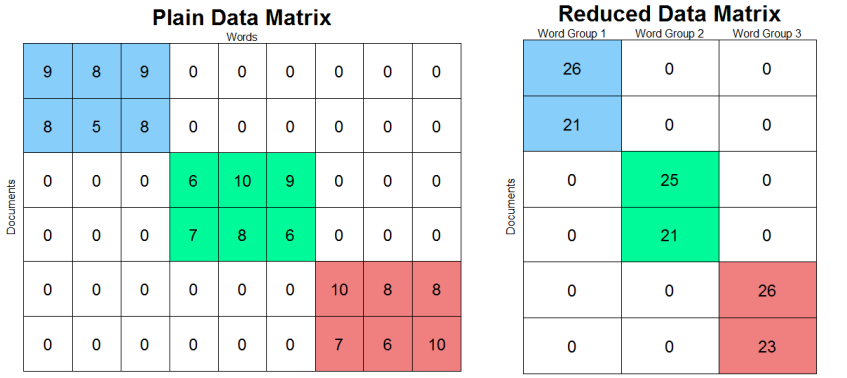


FIGURE 10.3: The left matrix shows a sample data matrix, color coded to highlight the obvious similarity between certain columns. The right matrix shows one way of grouping those columns together, by simply summing the values of all correlated columns into one single column with the sum total.

We never implemented this idea, so we can't say how well it might work. Of course, if the goal is to reduce runtime, then it's important that the time saved is not simply wasted on the feature clustering step instead. It's also uncertain the extent of which redundant or highly correlated columns detract from our clustering algorithm, if they hinder the algorithm at all. A lot of further exploration is required before implementation of this step.

10.4 Divisive Clustering (Cluster Selection)

Throughout our work on the 20 Newsgroup training datasets, we always had the ground truth knowledge of how many clusters we were looking for. In the Verizon data, and in many applications, we don't know the true amount of clusters to search for. Moreover, there often isn't one true number of clusters; rather, it is often an "open question" up to interpretation.

In prior research, we discovered that the NCut algorithm has been adapted to determine cluster membership iteratively, without requiring the initial knowledge of number of clusters. The algorithm essentially implements NCut in a divisive context, partitioning the dataset into two groups at a time until there is no effective partition left to be made.

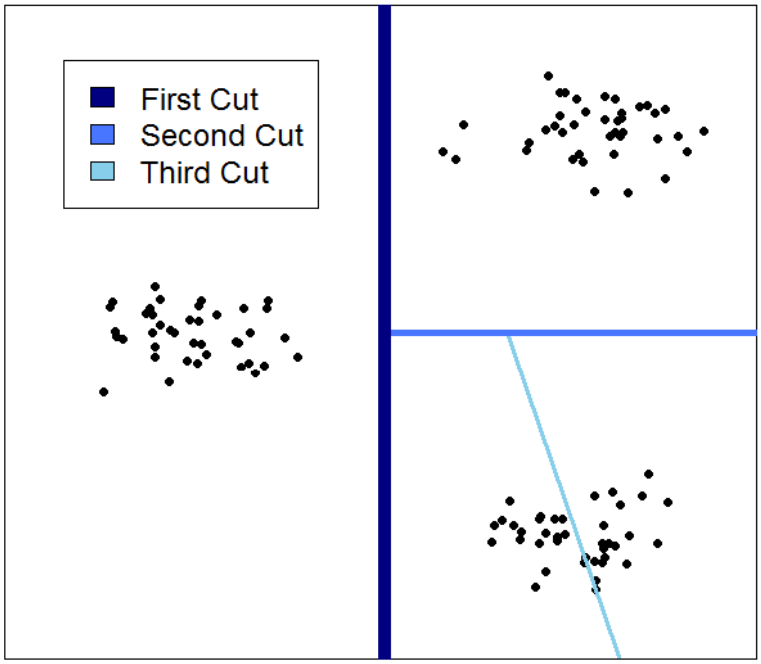


FIGURE 10.4: In this sample dataset, we showcase the procedure of divisive NCut. The first cut takes the cleanest partition of two data clusters. The second cutting stage then looks at those two resulting groups, and looks for any "good" cuts to be made within those groups. In this dataset, the left partition has no "good" cut to be made, while the right partition is once again split into two groups. Within the three new subgroups (the left cluster, and the two groups on the clusters), there is no "good" cut left to be made; notice how the third cut shown bottom–right would necessarily pass through strongly-connected graph edges (in terms of the mutual similarity between vertices). We therefore stop after two cuts, or three total clusters.

10.5 Categorical and Missing Data

In many applications, the data is a mix of numerical and categorical data. In our case, Verizon provided not only website frequency information but also user demographics, such as gender, location, browsing device (browser vs. app), ethnicity, and others. However, we did not incorporate this data into our algorithm, instead focusing on the website frequency data for our analysis. There is almost certainly some useful signal in the other data, but we avoided it partly because we did not have time to pivot our algorithm toward this extra information and partly because our training data (the 20 Newsgroup dataset) did not include any categorical information we could use in our algorithm construction and testing phases.

In general, one would need to find an approach to *balance* the information between demographic and categorical data. Simply concatenating the categorical and frequency data in our context would almost certainly be ineffective, as the amount of frequency columns would severely outnumber the amount of categorical columns. In the Verizon dataset, there are about 70 million unique website ID's, but only around 100 demographic variables. This would effectively drown out any signal

present in the categorical data. Rather, there would need to be careful implementation of column weighting and/or similarity measurement to reconcile the many differences between the numerical and categorical data.

Moreover, demographic data may introduce missing values into the data. In the Verizon dataset, we are given complete information of each web user's web browsing, but many people either opted out of or simply never provided their demographic information (such as gender, ethnicity, etc.). These values may be somewhat predictable by the content of the count data, but it still introduces new elements of variability and complexity into the data. A more careful reflection is desired on the intricacies of such data.

Bibliography

[1] Richard Bellman. Dynamic programming and lagrange multipliers. *Proceedings of the National Academy of Sciences*, 42(10):767–769, 1956.

[2] Ronald R Coifman and Stéphane Lafon. Diffusion maps. *Applied and computational harmonic analysis*, 21(1):5–30, 2006.

[3] Chris Ding and Xiaofeng He. K-means clustering via principal component analysis. In *Proceedings of the twenty-first international conference on Machine learning*, page 29. ACM, 2004.

[4] Susan Dumais, John Platt, David Heckerman, and Mehran Sahami. Inductive learning algorithms and representations for text categorization. In *Proceedings of the seventh international conference on Information and knowledge management*, pages 148–155. ACM, 1998.

[5] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[6] Przemysław Kazienko and Michał Adamski. Adrosa—adaptive personalization of web advertising. *Information Sciences*, 177(11):2269–2295, 2007.

[7] David J Ketchen Jr and Christopher L Shook. The application of cluster analysis in strategic management research: an analysis and critique. *Strategic management journal*, pages 441–458, 1996.

[8] Ken Lang. Newsweeder: Learning to filter netnews. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 331–339, 1995.

[9] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS*, pages 849–856. MIT Press, 2001.

[10] William M Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical association*, 66(336):846–850, 1971.

[11] Joshua Rosen. Pyspark internals.

[12] Hinrich Schütze. Introduction to information retrieval. In *Proceedings of the international communication of association for computing machinery conference*, 2008.

[13] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(8):888–905, August 2000.

[14] R Core Team. R language definition. *Vienna, Austria: R foundation for statistical computing*, 2000.

[15] Nguyen Xuan Vinh, Julien Epps, and James Bailey. Information theoretic measures for clusterings comparison: is a correction for chance necessary? In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1073–1080. ACM, 2009.

[16] Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.

Appendix A

R Packages and Codes

A.1 R Packages

Matrix
RSpectra
irlba
fclust
rgl
Rtsne
data.table
lattice

A.2 Main Function

```
1
2  mainfunction <- function(data, nclust,
3
4      # Data input type:
5      sparse=TRUE, convertsparse=TRUE,
6
7      # Saving and/or Returning the output:
8      save=TRUE, return=TRUE,
9
10     # Preset  argument combinations:
11     preset  =  0,
12
13     # Column weighting argument defaults:
14     weightfunction="IDF", binary=TRUE, lower=2, upper=NULL, par1=NULL,
           par2=NULL, mode=NULL,
15
16     # Similarity  function argument defaults:
17     simfunction = "cosine",  simscale=NULL,
18     rowscaling = NULL, colscaling = NULL, sigma = NULL, centers = NULL, seed
           = NULL, distance = NULL,
19
20     # Clustering function argument defaults:
21     clusterfunction="DiffusionMap", t=.5,  kmeans.method="kmeans", m=NULL,
22
23     # SVD options for weighted (IDF) data:
24     weight.SVD=FALSE, SVDdim=200, SVDprint=FALSE,
```

```
25      dim1=1, dim2=2, dim3=3, filepath=NULL, # for plotting 3D graphs
26      # specify filepath if you want to save as PDF
27
28      # SVD options for similarity (cosine) matrix:
29      SVDsim=TRUE, simdim=3, dim1sim=1, dim2sim=2, dim3sim=3,
            SVDsim.plot=FALSE, sim.filepath=NULL,
30
31      # Cluster Insights:
32      insights=FALSE, vocab=NULL, nfeatures=20, n.ins=NULL, insight.plot =
            TRUE, insight.filepath = NULL,
33      # insight.filepath should be a folder, not a filename, if you want to store
            multiple files
34
35
36      end.of.arguments=NULL) # end of arguments (for neatness)
37
38  {
39
40      # newfolder <- gsub(":", "_", paste ("~/",Sys.time(),sep="_"))
41      #
42      # dir.create(newfolder)
43      #
44      # setwd(newfolder)
45
46      require(Matrix)
47
48
49      #####
50      #####
51      ##### PRESET ARGUMENTS #####
52
53      if (preset==1) {
54
55          weight.SVD = TRUE
56          weightfunction = "IDF^2"
57      }
58
59      if (preset==2) {
60
61          kmeans.method="poly.fuzzy"
62      }
63
64      if (preset==3) {
65
66          clusterfunction="NJW"
67          kmeans.method="poly.fuzzy"
68      }
69
70      if (preset==4) {
71
72          weight.SVD = TRUE
```

```
73          weightfunction = "IDF^2"
74          kmeans.method="poly.fuzzy"
75      }
76
77      if (preset==5) {
78
79          weight.SVD = TRUE
80          weightfunction = "IDF^2"
81          clusterfunction="NJW"
82          kmeans.method="poly.fuzzy"
83      }
84
85
86      #####
87      #####
88      ##### DEFINE OUR FUNCTIONS #####
89
90      colweights <- function (data, weightfunction, sparseinput,
91          par1=NULL, par2=NULL, mode=NULL,
92          binary=TRUE, convertsparse=TRUE,
93          lower=2, upper=NULL) {
94
95          #####
96          ##### construct Matrix object for use with "Matrix" package #####
97
98          if (sparseinput==T) {     # given a sparse matrix -- convert to Matrix
                class
99
100             if (is.matrix(data)) {
101                 if (binary==T) {
102                     data <- sparseMatrix(i=data[,1], j=data[,2], x=rep(1,
                            nrow(data)))
103                 }
104
105                 else {
106                     data <- sparseMatrix(i=data[,1], j=data[,2], x=data[,3])
107                 }
108             }
109
110             else if (is.data.frame(data)) {
111
112                 if (binary==T) {
113                     data <- sparseMatrix(i=data[,1], j=data[,2], x=rep(1,
                            nrow(data)))
114                 }
115
116                 else {
117                     data <- sparseMatrix(i=data[,1], j=data[,2], x=data[,3])
118                 }
119             }
120
```

```
121             else if (binary==T) {
122
123                 data[data>0]<-1
124
125             }
126
127         }
128
129         else{    # given a dense matrix (sparseinput == F)
130
131             if (binary==TRUE) {
132
133                 data[data>0]<-1
134
135             }
136
137             if (convertsparse==TRUE) { # convert dense matrix to sparse matrix
138                 if (is.matrix(data)) {
139                     data <- Matrix(data, sparse=T)
140                 }
141
142                 else if (is.data.frame(data)) {
143                     data <- as.matrix(data)
144                     data <- Matrix(data)
145                 }
146             }
147
148             else {   # keep data in dense format, but convert to class = Matrix
149                 data <- as.matrix(data)
150                 data <- Matrix(data)
151             }
152
153         }
154
155
156
157         #####
158         #####
159         ##### Find density proportion of each column #####
160
161         weightfunction <- as.character(weightfunction)
162
163         if (binary==F) {
164
165             temp <- data
166             temp[temp>0]<-1
167             colsum <- colSums(temp)
168             colprop <- colsum/nrow(temp)
169
170         }
171
```

```
172         else {
173             colsum <- colSums(data)
174             colprop <- colsum/nrow(data)
175         }
176
177
178         #####
179         #####
180         ##### Remove columns outside your threshold (and monitor the rows)
                #####
181
182         if( !(is.null(lower))) {
183
184             data <- data[,which(colsum >= lower)] # colsum is the sum of
                    nonzero entries
185
186             colsum <- colsum[which(colsum >= lower)]
187
188         }
189
190         if( !(is.null(upper))) {
191
192             data <- data[,which(colsum <= upper)] # colsum is the sum of
                    nonzero entries
193
194             colsum <- colsum[which(colsum <= upper)]
195
196         }
197
198         rowsum <- rowSums(data)
199
200         if(min(rowsum) <= 0) { # some rows could lose all nonzero entries when
                you trim columns
201
202             # resp <- readline(prompt="One or more rows has zero weight. \n
203             #       Make sure that you fix this before continuing. \n
204             #       Press the ENTER key to continue. \n")
205
206             badrows <- which(rowsum<=0)
207
208             data <- data[-badrows,]
209
210             cat(length(badrows), " rows have zero weight, and will be removed.")
211
212
213
214         }
215
216
217
218
```

```
219         #####
220         #####
221         ##### Calculate column-weighted matrix & return #####
222
223         if (sparseinput==F & convertsparse==F) { # if you insist on using a
                dense matrix
224
225             if (weightfunction == "beta") {    # par1 = alpha, par2 = beta
226
227                 x <- seq(0,1, length=1000)
228                 mode.beta <- max(dbeta(x, shape1=par1, shape2=par2))
229
230                 colweights <- dbeta(colprop, shape1=par1, shape2=par2)
231                 colweights <- colweights/max(mode.beta) # scale to (0,1) range
232                 colweights <- sqrt(colweights)
233                 return(t(t(data)/colweights))
234
235             }
236
237             else if (weightfunction == "step") {      # par1 = min cutoff, par2 =
                    max cutoff
238
239                 return(data[,colprop > par1 & colprop < par2])
240
241             }
242
243             else if (weightfunction == "linear") {
244
245                 slope1 = 1/mode
246                 slope2 = -1/(1-mode)
247
248                 linweight <- function (density) {
249                     if (density < mode) { return(slope1*density) }
250                     else{return(slope2*(density-1) ) }
251                 }
252
253                 colweights <- sapply(colprop, linweight)
254                 colweights <- sqrt(colweights)
255                 return(t(t(data)/colweights))
256
257             }
258
259             else if (weightfunction == "IDF") {
260
261                 # IDF column weighting = log( N/ 1+density )
262                 data.idf <- log(nrow(data)/(1 + colsum))
263                 data.idf.diag <- Diagonal(n = length(data.idf), x=data.idf)
264
265                 # multiply each column by its IDF weight
266                 data. tfidf <- crossprod(t(data), data.idf.diag)
267                 return(data. tfidf )
```

```
268
269             # Row normalize
270             # data. tfidf .rn <- data.tfidf / sqrt(rowSums(data.tfidf^2))
271             # data. tfidf .rn <- data.tfidf / rowSums(data.tfidf)
272             # return(data. tfidf .rn)
273
274         }
275
276         else if (weightfunction == "IDF^2") {
277
278             # IDF column weighting = log( N/ 1+density )
279             data.idf <- (log(nrow(data)/(1 + colsum)))^2
280
281             # Multiply each column by its IDF weight
282             data.idf.diag <- Diagonal(n = length(data.idf), x=data.idf)
283             data. tfidf <- crossprod(t(data), data.idf.diag)
284             return(data. tfidf )
285
286             # Row normalize
287             # data. tfidf .rn <- data.tfidf / sqrt(rowSums(data.tfidf^2))
288             # data. tfidf .rn <- data.tfidf / rowSums(data.tfidf)
289             # return(data. tfidf .rn)
290
291         }
292
293         else if (weightfunction == "none") {
294
295             return(data)
296
297         }
298
299         else {stop("Pick a valid weight method.")}
300
301     }
302
303     else { # sparse matrix calculations
304
305         if (weightfunction == "beta") {    # par1 = alpha, par2 = beta
306
307             x <- seq(0,1, length=1000)
308             mode.beta <- max(dbeta(x, shape1=par1, shape2=par2))
309
310             colweights <- dbeta(colprop, shape1=par1, shape2=par2)
311             colweights <- colweights/max(mode.beta) # scale to (0,1) range
312             colweights <- sqrt(colweights)
313             return(t(t(data)/colweights))
314
315         }
316
317         else if (weightfunction == "step") {      # par1 = min cutoff, par2 =
                max cutoff
```

```
318
319             return(data[,colprop > par1 & colprop < par2])
320
321         }
322
323         else if (weightfunction == "linear") {
324
325             slope1 = 1/mode
326             slope2 = -1/(1-mode)
327
328             linweight <- function (density) {
329                 if (density < mode) { return(slope1*density) }
330                 else{return(slope2*(density-1) ) }
331             }
332
333             colweights <- sapply(colprop, linweight)
334             colweights <- sqrt(colweights)
335             return(t(t(data)/colweights))
336
337         }
338
339         else if (weightfunction == "IDF") {
340
341             # IDF column weighting = log( N/ density )
342             data.idf <- log(nrow(data)/(colsum))
343
344             # Multiply each column by its IDF weight
345             data.idf.diag <- Diagonal(n = length(data.idf), x=data.idf)
346             data.tfidf <- crossprod(t(data), data.idf.diag)
347             return(data.tfidf )
348
349             # Row normalize
350             # data.tfidf.rn <- data.tfidf/ sqrt(rowSums(data.tfidf^2))
351             # data.tfidf.rn <- data.tfidf/ rowSums(data.tfidf)
352             # return(data.tfidf.rn)
353
354         }
355
356         else if (weightfunction == "IDF^2") {
357
358             # IDF column weighting = log( N/ density )
359             data.idf <- (log(nrow(data)/(colsum)))^2
360
361             # Multiply each column by its IDF weight
362             data.idf.diag <- Diagonal(n = length(data.idf), x=data.idf)
363             data.tfidf <- crossprod(t(data), data.idf.diag)
364             return(data.tfidf )
365
366             # Row normalize
367             # data.tfidf.rn <- data.tfidf/ sqrt(rowSums(data.tfidf^2))
368             # data.tfidf.rn <- data.tfidf/ rowSums(data.tfidf)
```

```
369                 # return(data. tfidf .rn)
370
371             }
372
373             else  if  (weightfunction == "none") {
374
375                 return(data)
376
377             }
378
379             else  {stop("Pick a  valid  weight method.")}
380
381         }
382
383     }
384
385
386     similarity  <- function(data, method, rowscaling = NULL, colscaling = NULL,
387         sigma = NULL, centers = NULL, seed = NULL, distance = NULL,
388         sparse = T,  simscale)  {
389
390         #####
391         ##### Column scaling #####
392
393         if  (!(is.null(colscaling)))  {
394             if  (colscaling  == "standardize") {
395                 data <- apply(data, 2,  scale)  }
396
397             else  {stop("Pick a  valid  column scaling.")}
398
399         }
400
401         #####
402         #####
403         ##### Row scaling #####
404
405         if  (!(is.null(rowscaling)))  {
406
407             if  (rowscaling == "L2") {
408                 data <- data/sqrt(rowSums(data^2)) }
409
410             else  if  (rowscaling == "L1") {
411                 data <- data/rowSums(data) }
412
413             else  {stop("Pick a  valid  row scaling.")}
414
415         }
416
417         #####
418         #####
419         ##### Distance -> Gaussian similarity (if  applicable)  #####
```

```
420
421         if (method == "Gaussian") {
422
423             # Calculate a distance metric.
424
425             if (distance == "JSdivergence") {
426
427                 jsdiv <- function(P){
428                     nrows <- length(P[,1])
429                     ncols <- length(P[1,])
430                     D <- matrix(rep.int(0, nrows ** 2), nrow = nrows)
431                     P[is.nan(P)] <- 0
432                     for(i in 2:nrows){
433                         p.row <- P[i,]
434                         for(j in 1:i-1){
435                             q.row <- P[j,]
436                             m.row <- 1/2 * (p.row + q.row)
437                             D[i,j] <- D[j,i] <- (1/2 * sum(p.row *
                                    log(p.row/m.row), na.rm =TRUE) + 1/2 *
                                    sum(q.row * log(q.row/m.row), na.rm = TRUE))
438                         }
439                     }
440                     return(D)
441                 }
442
443                 dist <- jsdiv(data) }
444
445             else if (distance == "L2") {
446
447                 dist <- as.matrix(dist(data)) }
448
449             else if (distance == "L1") {
450
451                 dist <- as.matrix(dist(data, method = "manhattan")) }
452
453             else {stop("Pick a valid distance.")}
454
455
456             # Convert distance to similarity
457
458             if (is.null(sigma)) {stop("Choose a sigma value.")}
459
460             else { Similarity <- exp(-1 * dist^2 / (2*sigma)) }
461
462         }
463
464         #####
465         #####
466         ##### Compute the NxN similarity matrix and return #####
467
468         ###
```

```
469         ### dense matrix:
470         ###
471
472         if (sparse == F) {
473
474             if (is.null(centers)) {
475
476                 if (method == "correlation") {
477
478                     centeredcolumns <- t(t(data)-colMeans(data)) # center the
                            data by column
479
480                     rowvar <- rowSums(centeredcolumns^2) # store the
                            variances for each row (where colMeans=0)
481
482                     Cov.matrix <- tcrossprod(centeredcolumns) # calculate the
                            covariance matrix (dot product all rows)
483
484                     # corr = cov(x,y) / sqrt(var(x)var(y)
485                     Corr.true <- Cov.matrix/sqrt(rowvar)
486                     Similarity <- t(t(Corr.true)/sqrt(rowvar))
487
488                     # # scale the matrix to (0,1), excluding the diagonal
489                     # diag(Similarity) <- rep(0,nrow(Similarity))
490                     # Similarity <- (Similarity -
                            min(Similarity))/(range(Similarity)[2] -
                            range(Similarity)[1])
491                     # diag(Similarity) <- rep(0, nrow(Similarity))
492
493                     # Set any negative similarities equal to zero
494
495                     Similarity[Similarity<0] <- 0
496
497                 }
498
499                 else if (method == "corr.hack") {
500
501                     # center the data by column
502                     centeredcolumns <- t(t(data)-colMeans(data))
503                     # # store the variances for each row (where colMeans=0)
504                     # rowvar <- rowSums(centeredcolumns^2)
505                     # calculate the covariance matrix (dot product all rows)
506                     Cov.matrix <- tcrossprod(centeredcolumns)
507                     # calculate the length of each row (eventually scale by
                            row&col)
508                     cov.rowsums <- rowSums(Cov.matrix^2)
509                     Corr.hack <- Cov.matrix/sqrt(cov.rowsums)
510                     Similarity <- t(t(Corr.hack)/sqrt(cov.rowsums))
511
512                     # # scale the matrix to (0,1), excluding the diagonal
513                     # diag(Similarity) <- rep(0,nrow(Similarity))
```

```
514                 # Similarity <- (Similarity -
                        min(Similarity))/(range(Similarity)[2] -
                        range(Similarity)[1])
515                 # diag(Similarity) <- rep(0, nrow(Similarity))
516
517                 # Set any negative similarities equal to zero
518
519                 Similarity[Similarity<0] <- 0
520
521             }
522
523             else if (method == "cosine") {
524
525                 rowlength <- rowSums(data^2)
526                 Dot.prods <- tcrossprod(data)
527                 Cosines <- Dot.prods/sqrt(rowlength)
528                 Similarity <- t(t(Cosines)/sqrt(rowlength))
529                 # diag(Similarity) <- rep(0,nrow(Similarity))
530                 # Similarity <- (Similarity -
                        min(Similarity))/(range(Similarity)[2] -
                        range(Similarity)[1])
531                 # diag(Similarity) <- rep(0, nrow(Similarity))
532
533                 # Set any negative similarities equal to zero
534
535                 Similarity[Similarity<0] <- 0
536
537             }
538
539             else if (method == "dotproduct") {
540
541                 Similarity <- tcrossprod(data)
542                 # diag(Similarity) <- rep(0,nrow(Similarity))
543                 # Similarity <- (Similarity -
                        min(Similarity))/(range(Similarity)[2] -
                        range(Similarity)[1])
544                 # diag(Similarity) <- rep(0, nrow(Similarity))
545
546                 # Set any negative similarities equal to zero
547
548                 Similarity[Similarity<0] <- 0
549
550             }
551
552         }
553
554         #####
555         #####
556         ##### Random centers similarity matrix #####
557
558         else {
```

```
559
560             if (is.numeric(seed)) { set.seed(seed) }
561
562             if (method == "correlation") {
563
564                 centeredcolumns <- data-colMeans(data) # center the data
                        by column
565
566                 rowvar.full <- rowSums(centeredcolumns^2) # store the
                        variances for each row (where colMeans=0)
567
568                 # rcenters is rxN matrix, r = kcenters % of rows, sampled
                        randomly
569                 kcenters <- centers*nrow(data)
570                 rcenters <-
                        as.matrix(centeredcolumns[sample(nrow(data),kcenters,replace
                        = FALSE), ])
571
572                 rowvar.centers <- rowSums(rcenters^2)
573
574                 dotprod.centers <- tcrossprod(centeredcolumns,rcenters)
575
576                 # corr = cov(x,y) / sqrt(var(x)var(y)
577                 Corr.true <- dotprod.centers/sqrt(rowvar.full)
578                 Corr.centers <- t(t(Corr.true)/sqrt(rowvar.centers))
579
580                 Similarity <- tcrossprod(Corr.centers)
581
582                 # scale the matrix to (0,1), excluding the diagonal
583                 diag(Similarity) <- rep(0,nrow(Similarity))
584                 Similarity <- (Similarity -
                        min(Similarity))/(range(Similarity)[2] -
                        range(Similarity)[1])
585                 diag(Similarity) <- rep(1, nrow(Similarity))
586
587             }
588
589             else if (method == "corr.hack") {
590
591                 centeredcolumns <- data-colMeans(data) # center the data
                        by column
592
593                 # rcenters is rxN matrix, r = kcenters % of rows, sampled
                        randomly
594                 kcenters <- centers*nrow(data)
595                 rcenters <-
                        as.matrix(centeredcolumns[sample(nrow(data),kcenters,replace
                        = FALSE), ])
596
597                 dotprod.centers <- tcrossprod(centeredcolumns,rcenters)
598
```

```
599             rowlengths <- rowSums(dotprod.centers)
600             collengths <- colSums(dotprod.centers)
601
602             # hack = cov(x,y) / sqrt(length(x)length(y))
603             Corr.hack <- dotprod.centers/sqrt(rowlengths)
604             Corrhack.centers <- t(t(Corr.hack)/sqrt(collengths))
605
606             Similarity <- tcrossprod(Corrhack.centers)
607
608             # scale the matrix to (0,1), excluding the diagonal
609             diag(Similarity) <- rep(0,nrow(Similarity))
610             Similarity <- (Similarity -
                    min(Similarity))/(range(Similarity)[2] -
                    range(Similarity)[1])
611             diag(Similarity) <- rep(1, nrow(Similarity))
612
613         }
614
615         else if (method == "cosine") {
616
617             rowlengths.full <- rowSums(data^2) # store the variances for
                    each row (where colMeans=0)
618
619             kcenters <- centers*nrow(data)
620             rcenters <-
                    as.matrix(data[sample(nrow(data),kcenters,replace =
                    FALSE), ])
621
622             rowlengths.centers <- rowSums(rcenters^2)
623
624             dotprod.centers <- tcrossprod(data,rcenters)
625
626             # cos = <x,y> / sqrt(length(x)length(y))
627             Cosine <- dotprod.centers/sqrt(rowlengths.full)
628             Cosine.centers <- t(t(Cosine)/sqrt(rowlengths.centers))
629
630             Similarity <- tcrossprod(Cosine.centers)
631
632             # scale the matrix to (0,1), excluding the diagonal
633             diag(Similarity) <- rep(0,nrow(Similarity))
634             Similarity <- (Similarity -
                    min(Similarity))/(range(Similarity)[2] -
                    range(Similarity)[1])
635             diag(Similarity) <- rep(1, nrow(Similarity))
636
637         }
638
639         else if (method == "dotproduct") {
640
641             kcenters <- centers*nrow(data)
```

```
642                     rcenters <-
                            as.matrix(data[sample(nrow(data),kcenters,replace =
                            FALSE), ])
643
644                     dotprod.centers <- tcrossprod(data,rcenters)
645
646                     Similarity <- tcrossprod(dotprod.centers)
647
648                     # scale the matrix to (0,1), excluding the diagonal
649                     diag(Similarity) <- rep(0,nrow(Similarity))
650                     Similarity <- (Similarity -
                            min(Similarity))/(range(Similarity)[2] -
                            range(Similarity)[1])
651                     diag(Similarity) <- rep(1, nrow(Similarity))
652
653                 }
654
655             }
656
657             #####
658             #####
659
660         }
661
662         ###
663         ### sparse matrix:
664         ###
665
666         else {
667
668             if (is.null(centers)) {
669
670                 if (method == "correlation") {
671
672                     # centeredcolumns <- data
673                     #
674                     # rowvar <- rowSums(centeredcolumns^2) # store the
                            variances for each row (where colMeans=0)
675                     #
676                     # Cov.matrix <- tcrossprod(centeredcolumns) # calculate the
                            covariance matrix (dot product all rows)
677                     #
678                     # # corr = cov(x,y) / sqrt(var(x)var(y)
679                     # Corr.true <- Cov.matrix/sqrt(rowvar)
680                     # Similarity <- t(t(Corr.true)/sqrt(rowvar))
681                     #
682                     # # scale the matrix to (0,1), excluding the diagonal
683                     # diag(Similarity) <- rep(0,nrow(Similarity))
684                     # Similarity <- (Similarity -
                            min(Similarity))/(range(Similarity)[2] -
                            range(Similarity)[1])
```

```
685                 # diag(Similarity) <- rep(0, nrow(Similarity))
686
687                 stop("Correlation does not work on a sparse matrix. Try
                        using a dense matrix instead.")
688
689             }
690
691             else if (method == "corr.hack") {
692
693                 # centeredcolumns <- data
694                 # # # store the variances for each row (where colMeans=0)
695                 # # rowvar <- rowSums(centeredcolumns^2)
696                 # # calculate the covariance matrix (dot product all rows)
697                 # Cov.matrix <- tcrossprod(centeredcolumns)
698                 # # calculate the length of each row (eventually scale by
                        row&col)
699                 # cov.rowsums <- rowSums(Cov.matrix^2)
700                 # Corr.hack <- Cov.matrix/sqrt(cov.rowsums)
701                 # Similarity <- t(t(Corr.hack)/sqrt(cov.rowsums))
702                 # # scale the matrix to (0,1), excluding the diagonal
703                 # diag(Similarity) <- rep(0,nrow(Similarity))
704                 # Similarity <- (Similarity -
                        min(Similarity))/(range(Similarity)[2] -
                        range(Similarity)[1])
705                 # diag(Similarity) <- rep(0, nrow(Similarity))
706
707                 stop("Correlation does not work on a sparse matrix. Try
                        using a dense matrix instead.")
708
709             }
710
711             else if (method == "cosine") {
712
713                 rowlength <- rowSums(data^2)
714                 data <- data/sqrt(rowlength)   # cosine normalizes each
                        vector to unit length
715                 Similarity <- tcrossprod(data)
716
717                 if(is.null(simscale)) {
718
719                     if (weight.SVD==TRUE) {simscale="negative"}
720
721                     if (weight.SVD==FALSE) {simscale="0-1"}
722                 }
723
724                 if (simscale == "negative") {
725
726                     # Set any negative similarities equal to zero
727
728                     Similarity[Similarity<0] <- 0
729                     diag(Similarity) <- rep(0, nrow(Similarity))
```

```
730                     }
731
732                     else if (simscale == "0-1") {
733
734                         # rescale to (0,1) interval
735
736                         diag(Similarity) <- rep(0,nrow(Similarity))
737                         Similarity <- (Similarity -
                                min(Similarity))/(range(Similarity)[2] -
                                range(Similarity)[1])
738                         diag(Similarity) <- rep(0, nrow(Similarity))
739                     }
740
741                 }
742
743                 else if (method == "dotproduct") {
744
745                     Similarity <- tcrossprod(data)
746
747                     if(is.null(simscale)) {
748
749                         if (weight.SVD==TRUE) {simscale="negative"}
750
751                         if (weight.SVD==FALSE) {simscale="0-1"}
752                     }
753
754                     if (simscale == "negative") { # Set negative similarities to
                            zero
755
756                         Similarity[Similarity<0] <- 0
757                     }
758
759                     else if (simscale == "0-1") { # rescale to (0,1) interval
760
761                         diag(Similarity) <- rep(0,nrow(Similarity))
762                         Similarity <- (Similarity -
                                min(Similarity))/(range(Similarity)[2] -
                                range(Similarity)[1])
763                         diag(Similarity) <- rep(0, nrow(Similarity))
764                     }
765
766                 }
767
768             }
769
770             #####
771             #####
772             ##### Random centers similarity matrix #####
773
774             else {
775
```

```
776         if (is.numeric(seed)) { set.seed(seed) }
777
778         if (method == "correlation") {
779
780             centeredcolumns <- data-colMeans(data) # center the data
                    by column
781
782             rowvar.full <- rowSums(centeredcolumns^2) # store the
                    variances for each row (where colMeans=0)
783
784             # rcenters is rxN matrix, r = kcenters % of rows, sampled
                    randomly
785             kcenters <- centers*nrow(data)
786             rcenters <-
                    as.matrix(centeredcolumns[sample(nrow(data),kcenters,replace
                    = FALSE), ])
787
788             rowvar.centers <- rowSums(rcenters^2)
789
790             dotprod.centers <- tcrossprod(centeredcolumns,rcenters)
791
792             # corr = cov(x,y) / sqrt(var(x)var(y)
793             Corr.true <- dotprod.centers/sqrt(rowvar.full)
794             Corr.centers <- t(t(Corr.true)/sqrt(rowvar.centers))
795
796             Similarity <- tcrossprod(Corr.centers)
797
798             # scale the matrix to (0,1), excluding the diagonal
799             diag(Similarity) <- rep(0,nrow(Similarity))
800             Similarity <- (Similarity -
                    min(Similarity))/(range(Similarity)[2] -
                    range(Similarity)[1])
801             diag(Similarity) <- rep(1, nrow(Similarity))
802
803         }
804
805         else if (method == "corr.hack") {
806
807             centeredcolumns <- data-colMeans(data) # center the data
                    by column
808
809             # rcenters is rxN matrix, r = kcenters % of rows, sampled
                    randomly
810             kcenters <- centers*nrow(data)
811             rcenters <-
                    as.matrix(centeredcolumns[sample(nrow(data),kcenters,replace
                    = FALSE), ])
812
813             dotprod.centers <- tcrossprod(centeredcolumns,rcenters)
814
815             rowlengths <- rowSums(dotprod.centers)
```

```
816                 collengths <- colSums(dotprod.centers)
817
818                 # hack = cov(x,y) / sqrt(length(x)length(y))
819                 Corr.hack <- dotprod.centers/sqrt(rowlengths)
820                 Corrhack.centers <- t(t(Corr.hack)/sqrt(collengths))
821
822                 Similarity <- tcrossprod(Corrhack.centers)
823
824                 # scale the matrix to (0,1), excluding the diagonal
825                 diag(Similarity) <- rep(0,nrow(Similarity))
826                 Similarity <- (Similarity -
                        min(Similarity))/(range(Similarity)[2] -
                        range(Similarity)[1])
827                 diag(Similarity) <- rep(1, nrow(Similarity))
828
829             }
830
831             else if (method == "cosine") {
832
833                 rowlengths.full <- rowSums(data^2) # store the variances for
                        each row (where colMeans=0)
834
835                 kcenters <- centers*nrow(data)
836                 rcenters <-
                        as.matrix(data[sample(nrow(data),kcenters,replace =
                        FALSE), ])
837
838                 rowlengths.centers <- rowSums(rcenters^2)
839
840                 dotprod.centers <- tcrossprod(data,rcenters)
841
842                 # cos = <x,y> / sqrt(length(x)length(y))
843                 Cosine <- dotprod.centers/sqrt(rowlengths.full)
844                 Cosine.centers <- t(t(Cosine)/sqrt(rowlengths.centers))
845
846                 Similarity <- tcrossprod(Cosine.centers)
847
848                 # scale the matrix to (0,1), excluding the diagonal
849                 diag(Similarity) <- rep(0,nrow(Similarity))
850                 Similarity <- (Similarity -
                        min(Similarity))/(range(Similarity)[2] -
                        range(Similarity)[1])
851                 diag(Similarity) <- rep(1, nrow(Similarity))
852
853             }
854
855             else if (method == "dotproduct") {
856
857                 kcenters <- centers*nrow(data)
```

```
858                     rcenters <-
                            as.matrix(data[sample(nrow(data),kcenters,replace =
                            FALSE), ])
859
860                     dotprod.centers <- tcrossprod(data,rcenters)
861
862                     Similarity <- tcrossprod(dotprod.centers)
863
864                     # scale the matrix to (0,1), excluding the diagonal
865                     diag(Similarity) <- rep(0,nrow(Similarity))
866                     Similarity <- (Similarity -
                            min(Similarity))/(range(Similarity)[2] -
                            range(Similarity)[1])
867                     diag(Similarity) <- rep(1, nrow(Similarity))
868
869                 }
870
871             }
872
873             #####
874             #####
875
876         }
877
878         return(Similarity)
879
880     }
881
882
883     clustering <- function(Weights, k, method, t=NULL, sparse=T,
884         kmeans.method="kmeans", m=NULL) {
885
886         if (method == "NJW") {
887
888             D <- rowSums(Weights) # Degrees matrix #####
889
890             ###
891             ### Check: if row has zero similarity, problems arise
892             ###
893
894             if( min(D) <= 0) {
895
896                 resp <- readline(prompt="One of your similarity rows has zero
                        weight. Would you like to set
897                     a 1 on the diagonal of the similarity? Type Y or N \n")
898                 if (resp == "Y" | resp == "y") {
899                     n <- which(D == 0)
900                     D[n] <- 1
901                 }
902
903                 else { stop("One of your rows has zero weight.") }
```

```
904             }
905
906             #####
907             ##### Subspace Projection #####
908             #####
909
910             D <- Diagonal(n=nrow(Weights),(D^-.5))
911             Z <- D %*% Weights %*% D
912
913
914             if (kmeans.method=="RKM") {
915
916                 ###
917                 ### Define the RKM function
918                 ###
919
920                 RKM <- function (data, nclus, ndim, alpha = NULL, method =
                        "RKM", center = TRUE,
921                     scale = TRUE, rotation = "none", nstart = 10, smartStart =
                            NULL,
922                     seed = 1234) {
923
924                     require(ggplot2)
925                     require(dummies)
926                     require(grid)
927                     require(corpcor)
928
929                     ssq = function(a) {
930                         t(as.vector(c(as.matrix(a))))%*%as.vector(c(as.matrix(a)))
931                     }
932
933                     if (is.null(alpha) == TRUE) {
934                         if (method == "RKM") {
935                             alpha = 0.5
936                         }
937                         else if (method == "FKM") {
938                             alpha = 0
939                         }
940                     }
941                     odata = data
942                     data = scale(data, center = center, scale = scale)
943                     # data = data.matrix(data)
944                     n = dim(data)[1]
945                     m = dim(data)[2]
946                     conv = 1e-06
947                     func = {
948                     }
949                     index = {
950                     }
951                     AA = {
952                     }
```

```
953                 FF = {
954                 }
955                 YY = {
956                 }
957                 UU = {
958                 }
959
960                 require(irlba)
961
962                 for (run in c(1:nstart)) {
963                     if (is.null(smartStart)) {
964                         myseed = seed + run
965                         set.seed(myseed)
966                         randVec = matrix(ceiling(runif(n) * nclus), n, 1)
967                     }
968                     else {
969                         randVec = smartStart
970                     }
971                     U = dummy(randVec)
972                     P = U %*% pseudoinverse(t(U) %*% U) %*% t(U)
973
974
975                     # A = eigen(t(data) %*% ((1 - alpha) * P - (1 - 2 *
                            alpha) *
976                     #             diag(n)) %*% data)$vectors
977                     # A = A[, 1:ndim]
978
979
980                     testobj <- t(data) %*% ((1 - alpha) * P - (1 - 2 *
                            alpha) * diag(n)) %*% data
981
982                     A <- partial_eigen(x=testobj, n = ndim, symmetric =
                            TRUE)$vectors
983
984
985                     G = data %*% A
986                     Y = pseudoinverse(t(U) %*% U) %*% t(U) %*% G
987                     f = alpha * ssq(data - G %*% t(A)) + (1 - alpha) *
                            ssq(data %*%
988                                 A - U %*% Y)
989                     f = as.numeric(f)
990                     fold = f + 2 * conv * f
991                     iter = 0
992                     while (f < fold - conv * f) {
993                         fold = f
994                         iter = iter + 1
995                         outK = try(kmeans(G, centers = Y, nstart = 100),
996                             silent = T)
997                         if (is.list(outK) == FALSE) {
998                             outK = EmptyKmeans(G, centers = Y)
999                         }
```

```
1000                v = as.factor(outK$cluster)
1001                U = diag(nlevels(v))[v, ]
1002                P = U %*% pseudoinverse(t(U) %*% U) %*% t(U)
1003
1004                # A = eigen(t(data) %*% ((1 - alpha) * P - (1 - 2 *
1005                #           alpha) * diag(n)) %*% data)$vectors
1006                # A = A[, c(1:ndim)]
1007
1008                testobj <- t(data) %*% ((1 - alpha) * P - (1 - 2 *
                        alpha) * diag(n)) %*% data
1009
1010                A <- partial_eigen(x=testobj, n = ndim, symmetric =
                        TRUE)$vectors
1011
1012
1013                G = data %*% A
1014                Y = pseudoinverse(t(U) %*% U) %*% t(U) %*% G
1015                f = alpha * ssq(data - G %*% t(A)) + (1 - alpha) *
1016                    ssq(data %*% A - U %*% Y)
1017            }
1018            func[run] = f
1019            FF[[run]] = G
1020            AA[[run]] = A
1021            YY[[run]] = Y
1022            UU[[run]] = U
1023            cat("Just finished iteration ", run, "\n")
1024        }
1025        mi = which.min(func)
1026        U = UU[[mi]]
1027        cluID = apply(U, 1, which.max)
1028        csize = round((table(cluID)/sum(table(cluID))) * 100, digits
                = 2)
1029        aa = sort(csize, decreasing = TRUE)
1030        require(plyr)
1031        cluID = mapvalues(cluID, from = as.integer(names(aa)), to =
                as.integer(names(table(cluID))))
1032        centroid = YY[[mi]]
1033        centroid = centroid[as.integer(names(aa)), ]
1034        if (rotation == "varimax") {
1035            require(stats)
1036            AA[[mi]] = varimax(AA[[mi]])$loadings
1037            FF[[mi]] = data %*% AA[[mi]]
1038            centroid = pseudoinverse(t(U) %*% U) %*% t(U) %*%
                    FF[[mi]]
1039            centroid = centroid[as.integer(names(aa)), ]
1040        }
1041        else if (rotation == "promax") {
1042            AA[[mi]] = promax(AA[[mi]])$loadings[1:m, 1:ndim]
1043            FF[[mi]] = data %*% AA[[mi]]
1044            centroid = pseudoinverse(t(U) %*% U) %*% t(U) %*%
                    FF[[mi]]
```

```
1045                    centroid = centroid[as.integer(names(aa)), ]
1046                }
1047                out = list()
1048                mi = which.min(func)
1049                out$obscoord = FF[[mi]]
1050                rownames(out$obscoord) = rownames(data)
1051                out$attcoord = data.matrix(AA[[mi]])
1052                rownames(out$attcoord) = colnames(data)
1053                out$centroid = centroid
1054                names(cluID) = rownames(data)
1055                out$cluID = cluID
1056                out$criterion = func[mi]
1057                out$csize = round((table(cluID)/sum(table(cluID))) * 100,
1058                    digits = 1)
1059                out$odata = odata
1060                out$scale = scale
1061                out$center = center
1062                out$nstart = nstart
1063                class(out) = "cluspca"
1064                return(out)
1065            }
1066
1067
1068            ###
1069            ### Run RKM on the normalized Z matrix
1070            ###
1071
1072            cluster.out = RKM(Z, nclus=k, ndim=k, method = "RKM",
                    rotation = "varimax", nstart=10)
1073
1074
1075            return(cluster.out)
1076        }
1077
1078
1079        # RSpectra is efficient for dense matrices
1080
1081
1082        if (k=="eigenvalue") {
1083
1084            require(irlba)
1085
1086            k <- 20
1087
1088            EZ <- partial_eigen(x=Z, n = k, symmetric = TRUE)$value
1089
1090            print(EZ)
1091
1092            k <- readline(prompt="\n Here is a list of the first 20
                    eigenvalues.
                    Pick whichever eigenvalue appears best. \n")
1093
```

```
1094
1095            k <- as.numeric(k)
1096
1097        }
1098
1099
1100        if (sparse==F) {
1101            require(RSpectra)
1102            EZ <- eigs_sym(Z, k+1, 'LM')$vector
1103            EZ <- EZ[,1:k]
1104        }
1105
1106
1107        # irlba is efficient and accurate for sparse matrices
1108
1109        else {
1110            require(irlba)
1111            EZ <- partial_eigen(x=Z, n = k+1, symmetric = TRUE)$vectors
1112            EZ <- EZ[,1:k]
1113        }
1114
1115
1116        # U is the L2-normalized eigenspace
1117
1118        U <- EZ/sqrt(rowSums(EZ^2))
1119
1120
1121        #####
1122        ##### k-Mmeans in this normalized eigenspace:
1123        #####
1124        {
1125            # Regular k-means
1126            if (kmeans.method=="kmeans") {
1127                cluster.out <- kmeans(U, centers=k, nstart = 100)
1128            }
1129
1130            # Fuzzy k-means
1131            else if (kmeans.method=="fuzzy") {
1132                require(fclust)
1133                if (is.null(m)) {
1134                    m <- 2
1135                }
1136                cluster.out <- FKM(X=U,k=k, m=m, RS=10)
1137            }
1138
1139            # Polynomial fuzzy k-means
1140            else if (kmeans.method=="poly.fuzzy") {
1141                require(fclust)
1142                if (is.null(m)) {
1143                    m <- .5
1144                }
```

```
1145                    cluster.out <- FKM.pf(X=U,k=k, b=m, RS=10)
1146                }
1147
1148                else {stop("Pick a valid k-means method.")}
1149            }
1150
1151    }
1152
1153
1154        else if (method == "Ncut") {
1155
1156            n <- nrow(Weights)
1157            dvec_inv = 1/sqrt(rowSums(Weights))
1158            #W_tilde = Matrix(rep(dvec_inv,n), ncol=n) * Weights *
                    t(Matrix(rep(dvec_inv,n),ncol=n))
1159            W_tilde = Diagonal(n,dvec_inv) %*% Weights %*%
                    Diagonal(n,dvec_inv)
1160            W_tilde = (W_tilde+t(W_tilde))/2
1161
1162            # diag(dvec_inv) %*% Weights %*% diag(dvec_inv) ?
1163            # why the average part?
1164
1165            if (sparse==F) {
1166                require(RSpectra)
1167                EZ <- eigs_sym(W_tilde, k, 'LM')$vector }
1168            else {
1169                require(irlba)
1170                EZ <- partial_eigen(x=W_tilde, n = k, symmetric = TRUE)$vectors
1171            }
1172
1173            V <- EZ
1174            V = matrix(rep(dvec_inv,k-1), ncol = k-1) * V[,2:k]
1175            V = V / (matrix(rep(sqrt(rowSums(V^2)),k-1),ncol=k-1))
1176
1177
1178            if (kmeans.method=="kmeans") {
1179                cluster.out <- kmeans(V, centers=k, nstart = 100)
1180            }
1181
1182            else if (kmeans.method=="fuzzy") {
1183                require(fclust)
1184                if (is.null(m)) {
1185                    m <- 2
1186                }
1187                cluster.out <- FKM(X=V,k=k, m=m, RS=10)
1188            }
1189
1190            else if (kmeans.method=="poly.fuzzy") {
1191                require(fclust)
1192                if (is.null(m)) {
1193                    m <- .5
```

```
1194                }
1195                cluster.out <- FKM.pf(X=V,k=k, b=m, RS=10)
1196            }
1197
1198            else {stop("Pick a valid k-means method.")}
1199
1200        }
1201
1202
1203        else if (method == 'DiffusionMap'){
1204
1205            if (is.null(t)) {
1206
1207                stop("Specify a t value.") }
1208
1209            require(RSpectra)
1210
1211            n <- nrow(Weights)
1212            dvec_inv = 1/sqrt(rowSums(Weights))
1213            #W_tilde = matrix(rep(dvec_inv,n), ncol=n) * Weights *
                    t(matrix(rep(dvec_inv,n),ncol=n))
1214            W_tilde = Diagonal(n,dvec_inv) %*% Weights %*%
                    Diagonal(n,dvec_inv)
1215            W_tilde = (W_tilde+t(W_tilde))/2
1216
1217            if (kmeans.method=="RKM") {
1218
1219                ###
1220                ### Define the RKM function
1221                ###
1222
1223                RKM <- function (data, nclus, ndim, alpha = NULL, method =
                        "RKM", center = TRUE,
1224                    scale = TRUE, rotation = "none", nstart = 10, smartStart =
                            NULL,
1225                    seed = 1234) {
1226
1227                    require(ggplot2)
1228                    require(dummies)
1229                    require(grid)
1230                    require(corpcor)
1231
1232                    ssq = function(a) {
1233                        t(as.vector(c(as.matrix(a))))%*%as.vector(c(as.matrix(a)))
1234                    }
1235
1236                    if (is.null(alpha) == TRUE) {
1237                        if (method == "RKM") {
1238                            alpha = 0.5
1239                        }
1240                        else if (method == "FKM") {
```

```
            alpha = 0
        }
    }
    odata = data
    data = scale(data, center = center, scale = scale)
    # data = data.matrix(data)
    n = dim(data)[1]
    m = dim(data)[2]
    conv = 1e-06
    func = {
    }
    index = {
    }
    AA = {
    }
    FF = {
    }
    YY = {
    }
    UU = {
    }

    require(irlba)

    for (run in c(1:nstart)) {
        if (is.null(smartStart)) {
            myseed = seed + run
            set.seed(myseed)
            randVec = matrix(ceiling(runif(n) * nclus), n, 1)
        }
        else {
            randVec = smartStart
        }
        U = dummy(randVec)
        P = U %*% pseudoinverse(t(U) %*% U) %*% t(U)


        # A = eigen(t(data) %*% ((1 - alpha) * P - (1 - 2 *
            alpha) *
        #             diag(n)) %*% data)$vectors
        # A = A[, 1:ndim]


        testobj <- t(data) %*% ((1 - alpha) * P - (1 - 2 *
            alpha) * diag(n)) %*% data

        A <- partial_eigen(x=testobj, n = ndim, symmetric =
            TRUE)$vectors


        G = data %*% A
```

```
1289            Y = pseudoinverse(t(U) %*% U) %*% t(U) %*% G
1290            f = alpha * ssq(data - G %*% t(A)) + (1 - alpha) *
                    ssq(data %*%
1291                    A - U %*% Y)
1292            f = as.numeric(f)
1293            fold = f + 2 * conv * f
1294            iter = 0
1295            while (f < fold - conv * f) {
1296                fold = f
1297                iter = iter + 1
1298                outK = try(kmeans(G, centers = Y, nstart = 100),
1299                    silent = T)
1300                if (is.list(outK) == FALSE) {
1301                    outK = EmptyKmeans(G, centers = Y)
1302                }
1303                v = as.factor(outK$cluster)
1304                U = diag(nlevels(v))[v, ]
1305                P = U %*% pseudoinverse(t(U) %*% U) %*% t(U)
1306
1307                # A = eigen(t(data) %*% ((1 - alpha) * P - (1 - 2 *
1308                #         alpha) * diag(n)) %*% data)$vectors
1309                # A = A[, c(1:ndim)]
1310
1311                testobj <- t(data) %*% ((1 - alpha) * P - (1 - 2 *
                        alpha) * diag(n)) %*% data
1312
1313                A <- partial_eigen(x=testobj, n = ndim, symmetric =
                        TRUE)$vectors
1314
1315
1316                G = data %*% A
1317                Y = pseudoinverse(t(U) %*% U) %*% t(U) %*% G
1318                f = alpha * ssq(data - G %*% t(A)) + (1 - alpha) *
1319                    ssq(data %*% A - U %*% Y)
1320            }
1321            func[run] = f
1322            FF[[run]] = G
1323            AA[[run]] = A
1324            YY[[run]] = Y
1325            UU[[run]] = U
1326            cat("Just finished iteration ", run, "\n")
1327        }
1328        mi = which.min(func)
1329        U = UU[[mi]]
1330        cluID = apply(U, 1, which.max)
1331        csize = round((table(cluID)/sum(table(cluID))) * 100, digits
                = 2)
1332        aa = sort(csize, decreasing = TRUE)
1333        require(plyr)
1334        cluID = mapvalues(cluID, from = as.integer(names(aa)), to =
                as.integer(names(table(cluID))))
```

```
1335                    centroid = YY[[mi]]
1336                    centroid = centroid[as.integer(names(aa)), ]
1337                    if (rotation == "varimax") {
1338                        require(stats)
1339                        AA[[mi]] = varimax(AA[[mi]])$loadings
1340                        FF[[mi]] = data %*% AA[[mi]]
1341                        centroid = pseudoinverse(t(U) %*% U) %*% t(U) %*%
                                FF[[mi]]
1342                        centroid = centroid[as.integer(names(aa)), ]
1343                    }
1344                    else if (rotation == "promax") {
1345                        AA[[mi]] = promax(AA[[mi]])$loadings[1:m, 1:ndim]
1346                        FF[[mi]] = data %*% AA[[mi]]
1347                        centroid = pseudoinverse(t(U) %*% U) %*% t(U) %*%
                                FF[[mi]]
1348                        centroid = centroid[as.integer(names(aa)), ]
1349                    }
1350                    out = list()
1351                    mi = which.min(func)
1352                    out$obscoord = FF[[mi]]
1353                    rownames(out$obscoord) = rownames(data)
1354                    out$attcoord = data.matrix(AA[[mi]])
1355                    rownames(out$attcoord) = colnames(data)
1356                    out$centroid = centroid
1357                    names(cluID) = rownames(data)
1358                    out$cluID = cluID
1359                    out$criterion = func[mi]
1360                    out$csize = round((table(cluID)/sum(table(cluID))) * 100,
1361                        digits = 1)
1362                    out$odata = odata
1363                    out$scale = scale
1364                    out$center = center
1365                    out$nstart = nstart
1366                    class(out) = "cluspca"
1367                    return(out)
1368                }
1369
1370
1371                ###
1372                ### Run RKM on the normalized W_tilde matrix
1373                ###
1374
1375                cluster.out = RKM(W_tilde, nclus=k, ndim=k+1, method =
                        "RKM", rotation = "varimax", nstart=10)
1376
1377                return(cluster.out)
1378            }
1379
1380            require(irlba)
1381
1382            if (k=="eigenvalue") {
```

```
1383
1384            k <- 20
1385
1386            EZ <- partial_eigen(x=W_tilde, n = k, symmetric = TRUE)$value
1387
1388            print(EZ)
1389
1390            k <- readline(prompt="\n Here is a list of the first 20
                    eigenvalues.
1391                Pick whichever eigenvalue appears best. \n")
1392
1393            k <- as.numeric(k)
1394
1395        }
1396
1397        EV <- eigs_sym(W_tilde, k+1, 'LM')
1398        V <- EV$vector
1399        lambda <- EV$value
1400
1401        V_inv = 1/sqrt((rowSums(V[,2:(k+1)]^2)))
1402        V <- matrix(rep(V_inv,k), ncol=k) * V[,2:(k+1)]
1403        V = matrix(rep(dvec_inv,k), ncol = k)  * V
1404
1405        V <- abs(matrix(rep(lambda[2:(k+1)], each=n), ncol=k))^(t)* V
1406
1407        # V = (matrix(rep(lambda[2:(k)], each=n), ncol=k-1)^t )* V
1408
1409        # if ( (t%%1) != 0){
1410        #     V <- abs(matrix(rep(lambda[1:(k)], each=n), ncol=k)^(t))* V
1411        # }
1412        # else {
1413        #     V <- abs(matrix(rep(lambda[1:(k)], each=n), ncol=k)^(t))* V
1414        # }
1415
1416        # run kmeans in eigenspace:
1417
1418        if (kmeans.method=="kmeans") {
1419            cluster.out <- kmeans(V, centers=k, nstart = 100)
1420        }
1421
1422        else if (kmeans.method=="fuzzy") {
1423            require(fclust)
1424            if (is.null(m)) {
1425                m <- 2
1426            }
1427            cluster.out <- FKM(X=V,k=k, m=m, RS=10)
1428        }
1429
1430        else if (kmeans.method=="poly.fuzzy") {
1431            require(fclust)
1432            if (is.null(m)) {
```

```
                    m <- .5
                }
                cluster.out <- FKM.pf(X=V,k=k, b=m, RS=10)
            }

            else if (kmeans.method=="RKM") {
                require(clustrd)
                cluster.out = cluspca(V, nclus=k, ndim=k, method = "RKM",
                    rotation = "varimax", nstart=10)
            }

        }


        else {stop("Pick a valid clustering method.") }

    }


    #####
    #####
    ##### RUN OUR FUNCTIONS #####


    ### Store a copy of the data for summary statistics later ###

    copydata <- colweights(data, weightfunction="none", sparseinput=sparse,
        binary=T)


    ### Column weighting ###

    col.args = list(data=data, weightfunction=weightfunction,
        sparseinput=sparse,
        par1=par1, par2=par2, mode=mode,
        binary=binary, convertsparse=convertsparse,
        lower=lower, upper=upper)


    weighteddata <- do.call(colweights, col.args)


    cat("Column weighting is finished. \n")


    ### if we want to get insights later, we need to store a copy of the data

    if (insights==TRUE & weight.SVD==TRUE){weighteddata2 <-
        weighteddata}
```

```
### change "sparse" to true if you converted to sparse in colweights step:

if (convertsparse==T) {sparse=T}


### Do you want to calculate SVD on the weighted data? ###

svd.data = NULL

if (weight.SVD==TRUE) {

    require(irlba)
    all.svd <- irlba(weighteddata, SVDdim)
    svd.data <- weighteddata %*% all.svd$v

    if (SVDprint==TRUE) { # print the SVD results

        require(rgl)
        svdoutput <- plot3d(svd.data[,dim1], svd.data[,dim2],
            svd.data[,dim3], col="blue")

        if (!(is.null(filepath))) {

            snapshot3d(filepath)
        }
    }

    weighteddata <- svd.data

    sparse = F

    convertsparse = F

    cat("SVD is finished. \n")

}


### Similarity Matrix ###

sim.args = list(data=weighteddata, method=simfunction,
    rowscaling = rowscaling, colscaling = colscaling,
    sigma = sigma, centers = centers, seed = seed,
    distance = distance, sparse = sparse, simscale=simscale)

simdata <- do.call(similarity, sim.args)

diag(simdata) <- 0


cat("Similarity matrix is finished. \n")
```

```
1530
1531
1532    ### Do you want to plot the SVD of the Similarity matrix? ###
1533
1534    simdata.svd = NULL
1535
1536    if (SVDsim==TRUE) {
1537
1538        require(irlba)
1539        sim.svd <- irlba(simdata, simdim)
1540        simdata.svd <- simdata %*% sim.svd$v
1541
1542        if (SVDsim.plot==T) {
1543            require(rgl)
1544            svdoutput <- plot3d(simdata.svd[,dim1sim], simdata.svd[,dim2sim],
                    simdata.svd[,dim3sim], col="blue")
1545        }
1546
1547        if (!(is.null(sim.filepath))) {
1548
1549            snapshot3d(sim.filepath)
1550        }
1551
1552        cat("Similarity SVD is finished. \n")
1553
1554
1555    }
1556
1557
1558    ### Clustering step ###
1559
1560    clust.args = list(Weights=simdata, k=nclust, method=clusterfunction,
1561        t=t, sparse=sparse, kmeans.method=kmeans.method, m=m)
1562
1563
1564    clusterdata <- do.call(clustering, clust.args)
1565
1566
1567    cat("Clustering is finished. \n")
1568
1569
1570    #####
1571    #####
1572    ##### Cluster Insights #####
1573
1574    if (insights==TRUE & weight.SVD==TRUE){weighteddata <-
        weighteddata2}
1575
1576    insight.output = NULL
1577
1578    if (insights==TRUE) {
```

```
1579
1580        getInsights <- function(cluster, vocab, n,
1581            plot, file){
1582
1583            require(RSpectra)
1584            require( lattice )
1585            svd.out <- svds(cluster, 4)
1586            v <- svd.out$v #dim(v) 61066    4
1587
1588            b_clr <- c("steelblue", "darkred")
1589            key <- simpleKey(rectangles = TRUE, space = "top", points=FALSE,
1590                text=c("Positive", "Negative"))
1591            key$rectangles$col <- b_clr
1592
1593            v1_top <- order(abs(v[,1]), decreasing = T)[1:n]
1594            v1_top_po <- v1_top[which(v[,1][v1_top] > 0)] # positive values
1595            v1_top_ne <- v1_top[which(v[,1][v1_top] < 0)] # negative values
1596            v1_top_t <- c(v1_top_po,v1_top_ne)
1597            v1_topn <- v[,1][v1_top_t]
1598            v1_top.words <- as.matrix(vocab[v1_top_t])
1599
1600            if (plot) {
1601                b1 <- barchart(as.table(v1_topn),
1602                    main="First column",
1603                    horizontal=FALSE, col=ifelse(v1_topn > 0,
1604                        b_clr[1], b_clr[2]),
1605                    ylab="Impact value",
1606                    scales=list(x=list(rot=55, labels=v1_top.words, cex=0.9)),
1607                    key = key)
1608                if (!is.null( file )) {
1609                    png(file)
1610                    print(b1)
1611                    dev.off()
1612                } else {
1613                    print(b1)
1614                }
1615            }
1616
1617            return( list(magnitude = v1_topn, keywords =
                as.character(v1_top.words)))
1618
1619        }
1620
1621
1622        if (is.null(n.ins)) { n.ins <- nclust }
1623
1624        insight.output = list()
1625
1626        for (clustID in 1:n.ins) {
1627
```

```
1628            insight.output[[n.ins]] <-
                    getInsights(cluster=weighteddata[clusterdata$clus==clustID,],
                    vocab=vocab,
1629                n=nfeatures, plot=T, file=insight.filepath)
1630
1631        }
1632
1633        cat("Cluster insights is finished. \n")
1634
1635    }
1636
1637
1638
1639
1640    #####
1641    #####
1642    ##### Gather the Output #####
1643
1644    # Gather any statistics of interest:
1645
1646    summarystats <- list(
1647        dimensions=dim(copydata),
1648        colsum=colSums(copydata),
1649        rowsum=rowSums(copydata),
1650        density=nnzero(copydata),
1651        max(copydata)
1652    )
1653
1654
1655    # Remove any data from the output:
1656
1657    col.args <- col.args[-1]
1658    sim.args <- sim.args[-1]
1659    clust.args <- clust.args[-1]
1660
1661
1662    # Output a list of relevant objects:
1663
1664    output <- list(col.args=col.args, sim.args=sim.args, clust.args=clust.args,
1665        summarystats=summarystats, clusterdata=clusterdata,
1666        svd.data=svd.data, simdata.svd=simdata.svd,
                insight.output=insight.output
1667        )#[which(c(T,T,T,T,T, weight.SVD, SVDsim, insights))]
1668
1669    if (save==T) {save(output, file = gsub(":", "_", paste
        ("~/",Sys.time(),sep="_")) )}
1670
1671    if (return==T) {return(output)}
1672
1673    # test <- list(w,x,y,z)[which(c(a,b,c,d))]
```

Appendix B

R Main Function Documentation

Important Arguments

Mandatory Arguments

These inputs must always be provided; there are no default values.

- data: any matrix, data frame, or object from the Matrix package.
- nclust: how many clusters do you want to find? Integer, or set "divisive kmeans" for an adaptive method, or set "eigenvalue" to see the eigenvalues of the similarity matrix before choosing.

Data Input Type

The function works most efficiently with sparse matrices, but can work with full–size matrices if necessary.

- sparse: Is your data already in sparse format? Specify sparse = TRUE if your data is a Nx3 "long format" matrix, or a Sparse object from the Matrix package. Default: TRUE, but make sure that this is correct.
- convertsparse: If your data is a full dense matrix (i.e. you set sparse=FALSE), specify convertsparse = TRUE convert the matrix to a sparse object for maximum efficiency. Default: TRUE.

Returning and/or Saving Output

- save: save the output into your working file directory as a .R list object, with file name = Sys.time(). Default: TRUE.
  - The file name is the result of calling Sys.time(), to guarantee unique names. If you cannot find the file in your directory, search for the current date in Y-M-D format (ex: "2017-05-08")
- return: store the output into your local workspace and explore the results. Default: TRUE.

Cluster Insights

Explore the clusters that we found in the main algorithm, analyzing their Singular Value Decomposition (SVD).

- insights: TRUE if you want this information, or = FALSE otherwise. Default: FALSE.
- vocab: input the vector of column names (from your raw input data). This is required for any meaningful analysis, providing the interpretation for the important features.
- nfeatures: number of top features to display for each cluster. Default = 20.

Presets

Pre-designated combinations of input arguments, showcasing our empirically strongest combinations. Polynomial fuzzy kmeans is used to measure results, but is not required for functional performance (and will slow down the algorithm drastically, if speed is important). Any unspecified input is ran with the default value.

- preset = 1: SVD with IDF^2 column weighting.
- preset = 2: polynomial fuzzy kmeans.
- preset = 3: NJW spectral clustering with polynomial fuzzy kmeans.
- preset = 4: SVD with IDF^2 column weighting and polynomial fuzzy kmeans.
- preset = 5: SVD with IDF^2 column weighting, NJW clustering, and polynomial fuzzy kmeans.

Function Output

This function outputs a list object, containing many sub–lists and other output objects. This main output list returns the following objects:

- $col.args: a list of the input arguments used for column weighting (for future reference purposes).
- $sim.args: a list of the input arguments used for the similarity matrix calculation (for future reference purposes).
- $clust.args: a list of the input arguments used for the spectral clustering algorithm.
- $summarystats: some basic summary statistics from the data: dimensions, a list of column densities from each column, a list of row densities from each row, the number of nonzero entries in the matrix, and the overall maximum value from the dataset.
- $clusterdata: The kmeans output from the final step of spectral clustering. This is another list, containing cluster IDs, sum-squares information, and other basic kmeans output.

  - Output$clusterdata$cluster will give the final cluster ID's, where "Output" is the object returned by the mainfunction output (you may store this object under a different name).

- $SVD.data: the dimensionally-reduced data, if SVD is used.

  - The SVD–reduced data will include 200 SVD columns by default. This can be changed with input argument SVDdim = ____.

- $simdata.svd: A dimensionally-reduced version of the similarity matrix. Only returned if SVDsim=TRUE.

- $insight.output: The various statistics and summary graphics about the content of each cluster. Only returned if insights=TRUE and if the "vocab" (a vector containing the true column names) is supplied.

Examples

1. Output <- mainfunction(data=Data, ncluster=5)

   - Data is a Nx3 long format matrix or a sparse object from the Matrix package
   - ncluster = 5 runs the algorithm for 5 clusters
   - Output will be a list of various objects, which can be referenced by the $ command
   - Output is automatically saved into your computer/server directory, with name equal to the specific time on your machine given by Sys.time() (ex: file name "2017-05-08 10:00:00").
   - All other arguments are set to their default values, described below.

2. Output <- mainfunction(data=Data, ncluster=5, sparse=FALSE, convertsparse=TRUE)

   - Same specifications as example 1, except the input object "Data" would be a dense matrix in this case rather than a sparse matrix (as noted by the "sparse=FALSE" argument).
   - "convertsparse=TRUE" specifies that the data will be converted to a Sparse object (as found in the Matrix package) for maximal efficiency.

3. mainfunction(data=Data, ncluster=5, preset=1, return=FALSE)

   - "preset=1" will run the function with preset arguments, specified in section B. This option could take any integer value 1 through 5.
   - "return=FALSE" prevents the function from printing any output or storing it in your workspace. Notice how we are not storing the result into any variable name. Instead, the function will save the output as a list object in your main file directory (as described in example 1) without printing any output.

4. Output <- mainfunction(data=Data, ncluster=5, insights=TRUE, vocab=website.names)

- "insights=TRUE" will run our cluster insights function, which identifies the defining variables for each cluster.
- The input argument for vocab must be a vector of column names, which I represented with a dummy variable "website.names".

Other Optional Arguments

Colweights Function

Here, we specify all of the pre-processing steps. The main features are:

- weightfunction: method for column weighting. Options: "IDF", "IDF$\hat{2}$", "beta", "step", "linear", "none". Default: "IDF", also try "IDF$\hat{2}$" for more aggressive weighting.
- binary: specify "binary=TRUE" if you want to convert your data to binary, or if your data is already binary. Default: TRUE.

Similarity Function

Compute the NxN similarity matrix. The main features are:

- simfunction: specify the method for measuring pairwise similarity. Options: "cosine", "Gaussian", "correlation", "dotproduct". Default: "cosine".

Clustering Function

Run a spectral clustering algorithm to partition the data into clusters.

- clusterfunction: choose a spectral clustering algorithm. Options: "DiffusionMap", "NJW", "Ncut". Default: "DiffusionMap".
- t: number of 'steps' to calculate in the diffusion mapping (if "DiffusionMap" is chosen as the clustering algorithm). Larger values give bolder clusters, but seem overaggressive in practice. Default = .5.
- kmeans.method: kmeans variations for subspace clustering. Options: "kmeans", "fuzzy" (fuzzy kmeans, with probabilistic clustering), "poly.fuzzy" (fuzzy kmeans, but with a polynomial fuzzifier). Warning: both types of fuzzy clustering are VERY slow. Default: "kmeans".

SVD Options for Weighted Data

Run Singular Value Decomposition on the weighted data matrix for dimension reduction (before calculating the similarity matrix).

- weight.svd: set =TRUE to run SVD, if desired. It appears that SVD dimension reduction may eliminate noise and improve results. Default: FALSE.
- SVDdim: number of dimensions to project with SVD. We have achieved maximal results by using around 100–200 SVD dimensions. Default: 200.

SVD Options for Similarity Matrix

Strictly for visualization purposes on the similarity matrix; not actually used in the clustering partition.

- SVDsim: set =TRUE if you want to output the visualization plots. Default: TRUE .