The goal of this project is to apply different clustering algorithms on textual data. We have used the famous 20-newsgroups dataset for this project. We calculated the top 10 categories based on the number of documents per category and converted them into bag-of-word models to fed into our clustering algorithms.

Data Collection

The famous 20-newsgroups dataset is available in online. It can also be found through python sklearn packages. For easy processing, I have used the package for obtaining the data. This dataset comprises around 18000 newsgroups on 20 topics. I sorted the number to get top 10 categories name and collected the data belonging to these top categories. In total, my final dataset contains 9917 documents. The distribution of documents according to their topic is as follows (Figure 1):

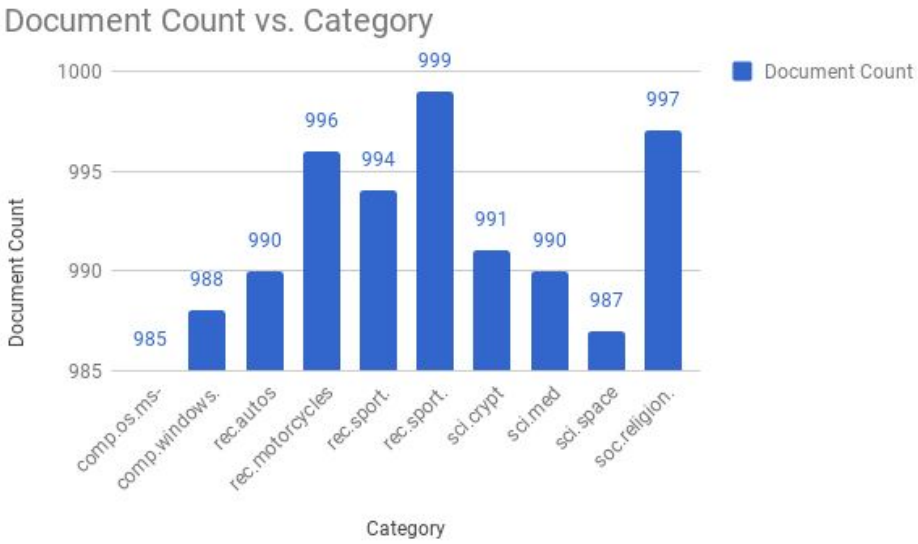

Figure 1: Distribution of Newsgroup data over top 10 categories

Word Vectorization

We need to convert the text into vectors to fed into the algorithms. There are some ways to do it. The most familiar ones are Count vectorization and TFIDF vectorization. To compare their performance I applied k-means algorithm (from sklearn package) on the vectorized data built from both vectorizers. Analyzing the SSE score reveals that TFIDF vectorization works better than the Count Vectorization (Figure 2)

Experiment Setup: max_features = 100000, min_df = 2, max_df =0.5, max_iteration = 100, k = 2,4,6,8,10,12,14,16

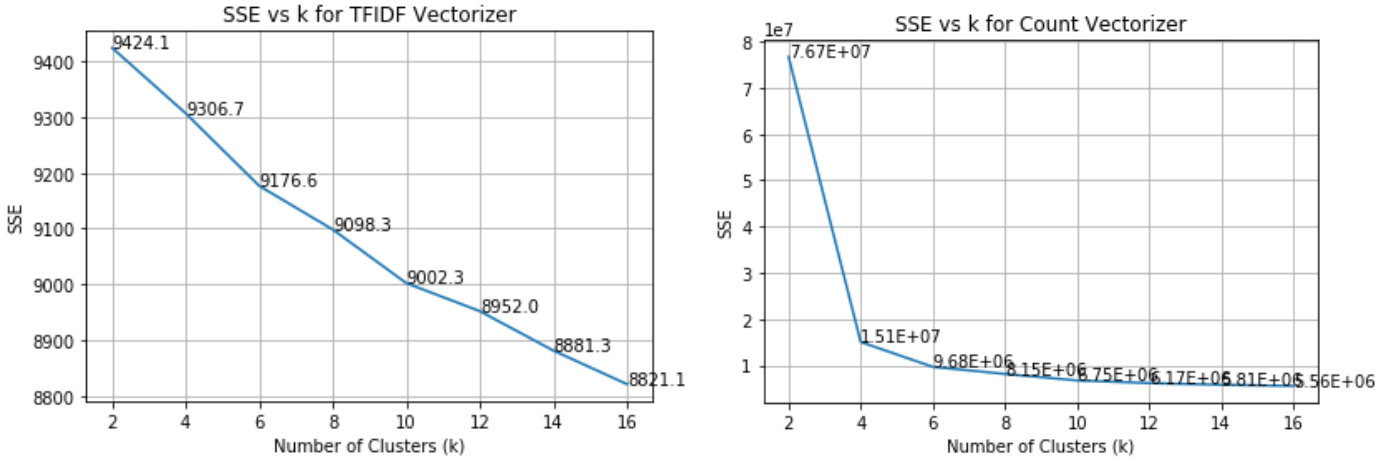

Figure 2: Comparison between Count Vectorizer and TFIDF Vectorizer

Clustering Algorithms

I applied three different clustering algorithms - 1) k-Means, 2) Bisecting k-Means and 3) DBSCAN.To compare the performance of the algorithms I calculated the SSE and average weighted entropy for each clustering results returned by the algorithms. As an input, I used TFIDF vectorization of the text documents. I didn't apply any text preprocessing techniques. To reduce the computational loads, I used max_features as 1000 for the TFDIF vectorization.

k-Means

I implemented my own k-Means algorithm using python's built-in libraries like numpy, math etc. To calculate the distance between the centroid and data points I used simple euclidean distance methods. I converted the sparse matrix returned by the vectorization process into ndarray to process them more easily. At the initial step the centroids are chosen randomly using using numpy's random function. And then the clusters were recalculated iteratively. I used max_iter = 100 considering the computation time and result variations. To compare the clustering performance, I have implemented functions to calculate SSE and entropy. For calculating SSE, I used euclidean distance. For entropy calculation, I generated the class distribution of the data for each cluster and then calculated the entropy for each cluster from the probability. And then I took the weighted average of the entropies of all the clusters. Keeping the other parameters unchanged, I set different values of k (form 2 to 16) to observe the performance of clustering over k's value. Figure 3 shows the comparison of the SSE and entropy value over different k.

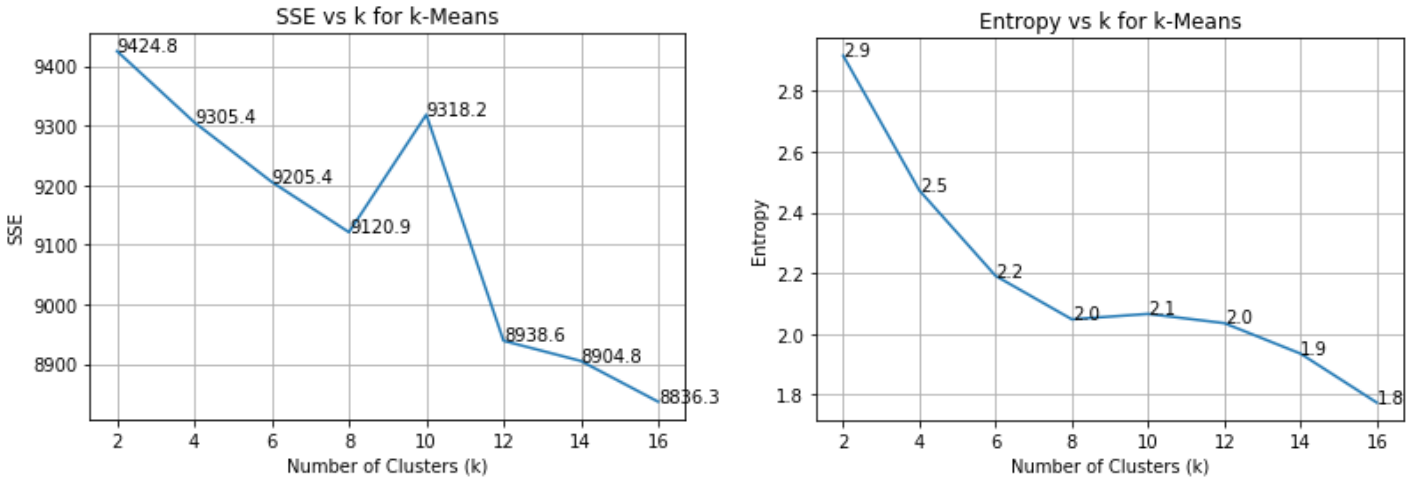

Figure 3: SSE and Entropy for k-Means Algorithm

Bisecting k-Means

For basic k-Means which we need to bisecting the clusters, I chose my own developed k-Means algorithm. After each iteration I sorted the clusters based on SSE score, and the cluster with largest SSE is selected for further bi-section. Like k-Means algorithm, I also calculated the SSE and entropy for the clustering results and plotted them over k. Figure 4 shows the result.

DBSCAN

I used sklearn package to use the dbscan algorithm in this part.In this part, I generated word vector using TFIDF with max_features = 10000. I just tuned the eps and min_samples parameters to observe the clustering performance. I chose 5 eps value (0.5, 0.7, 1.0, 1.3, 1.5) and 5 min_sample values for this experiment. In first step, I fixed the min_sample value to 10 and changed the eps value. I changed the min_sample value(10, 15, 20, 25, 40) in second step while I kept the eps value fixed to 1.0. Figure 5 shows the result.

Observation

k-Means shows more consistent result than Bisecting k-Means. We can see some ups and downs with the increase of k's value both for sse and entropy for Bisecting k-Means. For the original value of k (=10, for 10 categories), k-Means shows less SSE. But I can't say k_means is better here because I set different iterations

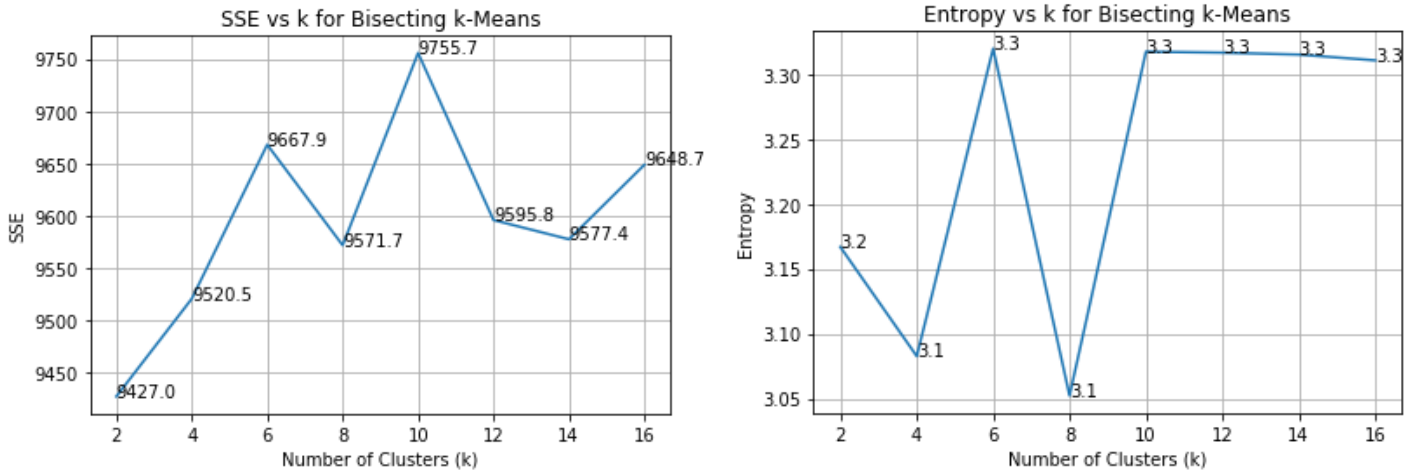

Figure 4: SSE and Entropy for Bisecting k-Means Algorithm

number for them(for k-Means 100, for bisecting 30). But surely, in terms of quality of the clustering bisecting k-Means outperforms other algorithms. For k >= 10, it shows higher entropy with constant rate, but the entropy of k-Means drops with the increase of k, although the SSE also falls. This means, with the increase of k, k-Means make the clustering cohesive, but in terms of homogeneity, the quality of the cluster suffers. The inconsistency of SSE for the bisecting k-Means may be due to low iteration numbers. Increasing iteration number for bisecting k-means may produce good results with lower SSE.

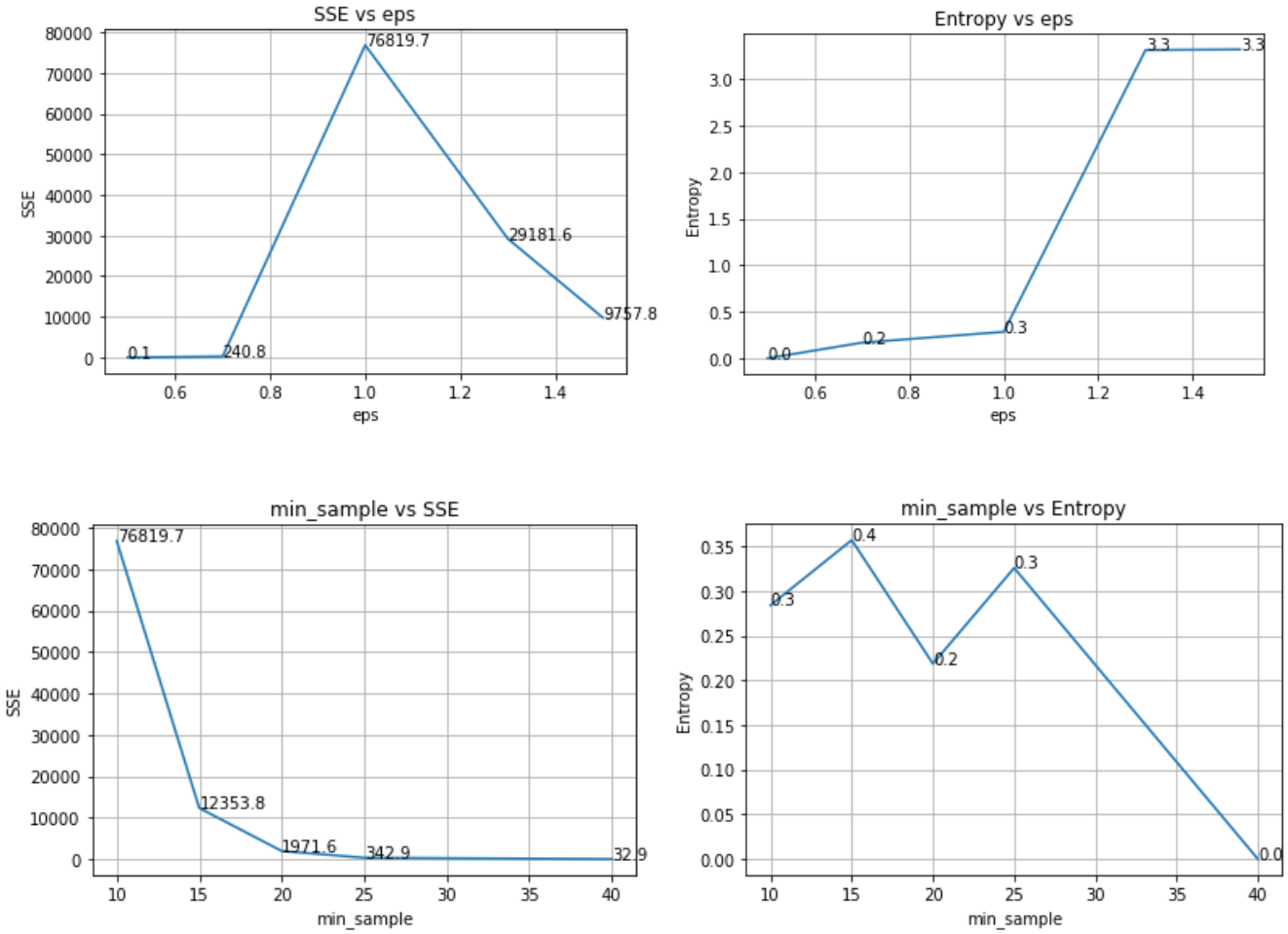

Figure 5: SSE and Entropy for DBSACN when min_sample is constant(1st row), when eps constant(2nd row)

For DBSCAN, when min_sample is constant, it seems lower eps produces cohesive clustering based on SSE, but it ruled out maximum data points as outliers. eps=1 produced highest SSE and then SSE got reduced with the increase of eps, but entropy became stable(k>=10). Entropy increases consistently, that means quality of the clusters in terms of homogeneity also increases and it produces good clustering.
So, increasing eps increased the number of clusters and reduced SSE here but achieved almost constant value for entropy. When we keep eps stable(=1.0) and varied min_sample, SSE got reduced, because the number of clusters and also data points in each clusters reduced, which affects the entropy also. So, higher min_sample doesn't produce good clustering. So, DBSCAN can produce good clustering in terms of both external and internal evaluation if the parameters can be tuned carefully.

Bonus Part

Word2Vec

Apart from Count Vectorization and TFIDF Vectorization, I tried another vectorization techniques, Word2Vec, which is more popular now due its capability to capture the context of the text. After preprocessing the text documents (tokenizing, cleaning punctuations, removing stop words, stemming), I used the gensim package to convert them into word2vec model. And then I used the model to get the word vector for each word of the each documents and took the average of the vectors for getting the vector for each document. I applied my own built k-Means and bisecting k-Means algorithm over the vector and followed the same process as before. Figure 6 shows the performance of this procedure in terms of SSE and entropy. For this experiment, I set vector dimension to 300, set window size to 5 and min_count to 1. Although, I was expecting a better performance than TFIDF, but result shows the opposite. A better tuning may produce a good result.

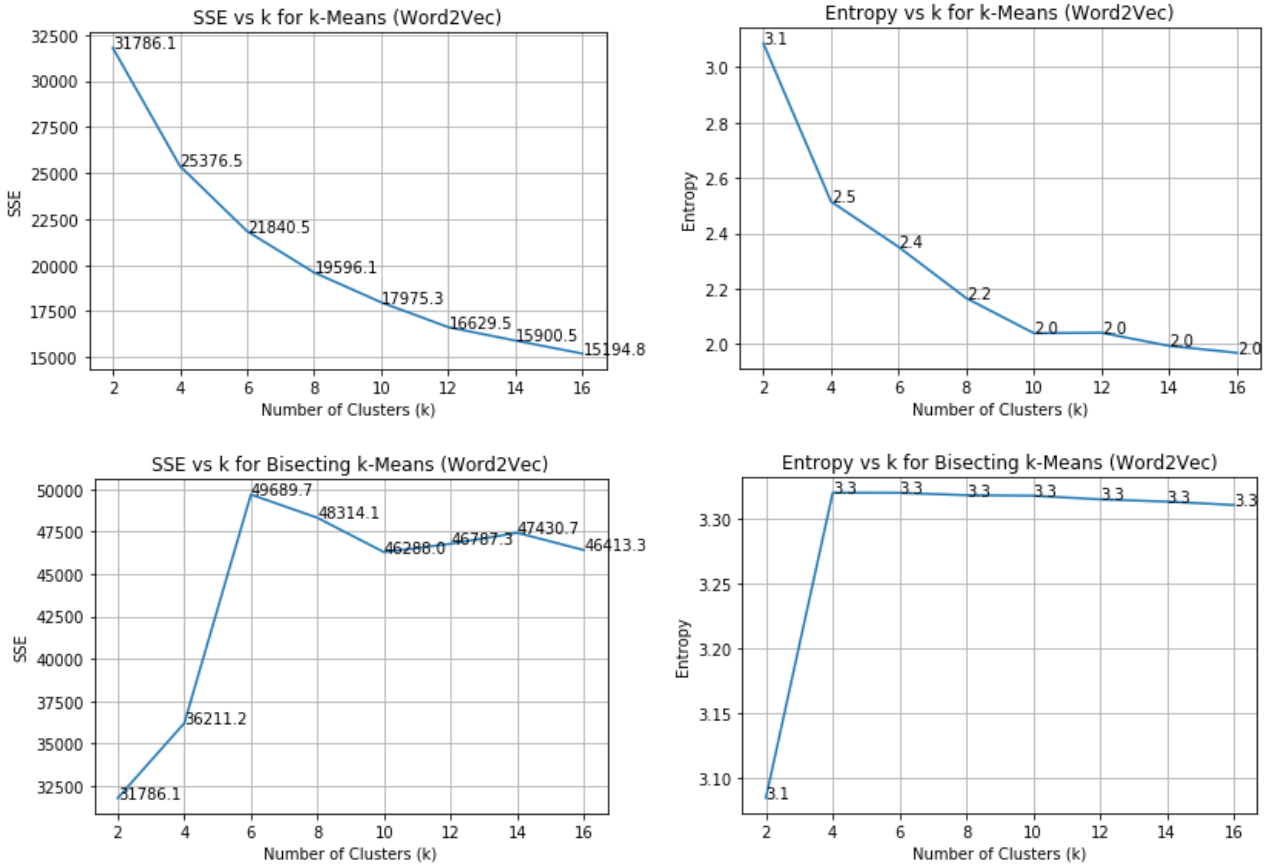

Figure 6: k-Means and Bisecting k-Means performance for Word2Vec model

DBSCAN

I also implemented DBSCAN algorithm using common python packages like numpy and math. But I didn't use this for the previous experiment as it is much slower. I just tried with one combination of eps and min_sample.