Lab Assignment 3 – Clustering

Daniel Marchese, Peter Jacobs

Introduction

The purpose of this assignment was to gain experience using various clustering techniques that have been discussed in lecture. As a group, we decided to implement two of these algorithms from scratch (DBSCAN and K-Means)

Vector Pre-Processing

In both of the clustering algorithms, the vectors were modified slightly to suit the implementations. First, all null-vectors were removed from the dataset since they can break the cosine-distance metric calculations. Second, we removed all of the vectors from the dataset with a topic of 'NULL'. Finally, vectors with multiple topics were duplicated so that each vector only contained one topic.

Entropy

Entropy was formulated according to equation (1)

$$\sum_{k=1}^{K} \frac{|C_k|}{N} \sum_{l \in C_k} \frac{|l|}{|C_k|} \log_{|L(C_k)|} \left(\frac{|C_k|}{|l|} \right) \quad \textbf{(1)}$$

Where K is the number of clusters, C_k is Cluster k, N is the size of the dataset, l is the set of all points with label l, and $L(C_k)$ is the set of all class labels in cluster C_k. This formulation allows the maximum entropy of a clustering to be 1.0 as opposed to an arbitrary value that would be received if using a base of 2 in the logarithm.

DBSCAN

The DBSCAN Algorithm was implemented as described in Data Mining Analysis and Concepts by M. Zaki and W. Meira. The provided source code contains a driver program that runs DBSCAN twice: once using Euclidean distance as the distance metric, and once using cosine distance as the metric.

Scalability and Performance

In DBSCAN, the main performance cost is in determining the quantity and identity of the epsilon neighbors of a given point (Zaki, Meira). In our implementation, this process involves (n choose 2) distance computations. Because this number of computations is asymptotically n^2, an important focus of the implementation was to calculate distance in an efficient manner.

Initially, we implemented our own distance metrics. The estimated time to complete DBSCAN on the entire data set with Euclidian distance with our implementation was over 7 hours, with 256 features. (Because attempting to run DBSCAN with 1000 features would have taken far longer than this, we initially focused only on the data set with 256 features. Following adjustments to improve performance, we did not have time to test with 1000 features. Therefore, throughout the DBSCAN analysis, timings and entropies refer to a 256 feature analysis.)

Library Distance Calculations

The first step taken in improving efficiency was the use of numpy and scipy to calculate Euclidean and cosine distance respectively. This step reduced the time to complete the n choose 2 distance computations by approximately 6 hours.

Given the necessity to be able to play with epsilon and minPts for analysis, we felt that a 1-hour runtime (and likely a longer one when using cosine), still needed improvement.

Sampling

The second step taken to improve performance was to sample the dataset so that less data had to be worked with. Specifically, 14% of the data set is chosen at random. A clustering is found for this subset of the data, centroids are found for each cluster (decimal values for features in the centroid are rounded to nearest whole number, which better facilitates cosine calculations), and the remaining 86% of of the data, record by record, is assigned to the cluster whose centroid each record is nearest to.

This centroid based assignment approach is efficient. However, it is prone to various problems that will be discussed in the quality section.

After these major adjustments, at what we felt were reasonable MinPts and Eps values (discussed later), the following timings were obtained:

Distance Measure	4 Run average Time	Min Pts	Epsilon
Cosine	140.80 seconds	128	.4
Euclidian	56.31 seconds	128	3.0

Clustering Quality and Results

DBSCAN clustering results are heavily influenced by the selection of minPts and epsilon.

Based on looking at typical distances between records for both Euclidian distance and cosine distance, we decided initially on the following epsilon values

Distance Measure	Initial Epsilon Value
Euclidian	5.5
Cosine	.12

The epsilon values were selected with the initial goal of finding a medium between too many small clusters, and one large cluster.

Initial minPts is approximately: 128 (5% of the 14% of data used for clustering)

Euclidian Parameter Adjustments

Under these initial conditions, the Euclidian DBSCAN reached the maximum depth of recursion. This indicated to us that under these parameters, there were way too many core points and many of them were nearby each other (hence leading to at least one very long density connected chain of points). Therefore, we decided to lower epsilon and raise minPts.

Raising minPts did not solve the problem until minPts=900! Clearly, our epsilon value was too high. Therefore, we adjusted our strategy and first lowered epsilon. This quickly solved the problem. The values for epsilon and minPts are listed in the table below.

Cosine Parameter Adjustments

Under the initial parameters (MinPts=128 & Epsilon=.12), all points were noise points. There were no core points (and hence no border points). Therefore, epsilon was raised until core points were discovered. The values for epsilon and minPts are listed in the table below.

*The parameter values reported in this table likely do not represent the optimal clustering, but they do avoid maximum depth recursion and a clustering with zero clusters (all noise points).

Distance Measure	4 Run average Entropy	Min Pts	Epsilon	4 Run Average # Clusters	4 Run Average # Noise Points	4 Run Variance for the 1 Cluster
Cosine	.507	128	.4	1	2071	31.31
Euclidian	.502	128	3.0	1	1610	27.21

The difference in entropy between the two different distance metrics does not appear to be significant. The .5 value of entropy observed is likely very close to the entropy of the entire data set. Because only one cluster was produced, aside from excluding noise points, DBSCAN in this case is futile.

Other Observations

After a great deal of trial and error, a clustering was found that yielded more than one cluster. However, it did not decrease entropy and it is not reported here.

Struggles to produce more than one cluster highlight a deficiency with DBSCAN: DBSCAN may tend towards one cluster. However, we cannot rule out that the data has just one natural cluster.

It is also important to note the effect of the cluster assignment strategy employed. Because DBSCAN clusters are not necessarily globular, selecting the representative of a cluster as its mean point is not a good strategy (even though it is efficient).

Also, in the case where more than one cluster is produced in the clustering, this implementation tends to put all records in the cluster with the smallest number of records, which is a major problem. This is has to do with the sparsity of the data, and is not elaborated further.

K-Means

The implementation of K-Means in this project does not follow any specific formulation of the problem. It is a canonical implementation using pure Python which halts iteration once identical clusters are produced in adjacent iterations. As will be discussed later, this implementation choice lead to a fairly efficient implementation.

Scalability and Performance

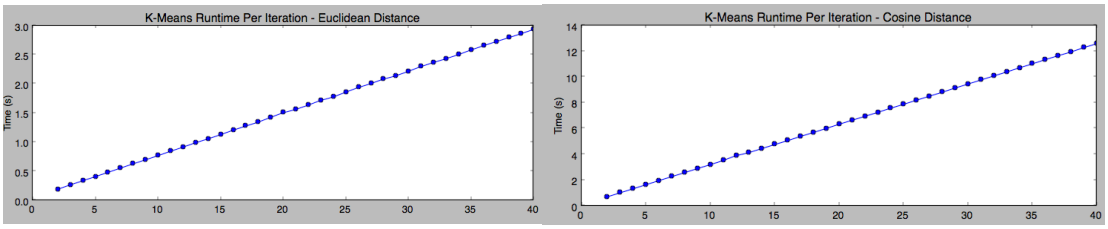
Distance Calculations

As was learned during the implementation of DBSCAN, the calculation of distance is a true bottleneck in most clustering processes, and this is especially so since the distances can't be cached ahead of time in a distance matrix. With this in mind, we used numerics libraries for our

distance metric calculations to save on time. For calculating Euclidean distance, we used numpy's `linalg.norm` module; for calculating cosine distance we used scipy's `spatial.distance.cosine` function (which leverages the numpy data structures).

Performance Results

On the full set of 1000-dimensional vectors, the average time per iteration of K-means was calculated and is visualized in the graphs below.



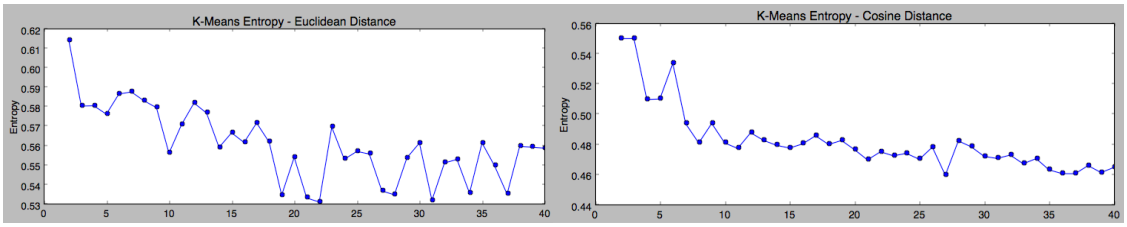
These results are not surprising since K-Means is a linear-time algorithm (usually). A single pass over the dataset requires $K * N$ distance calculations, so it makes sense that the time to organize the data into clusters would increase linearly with K. It should be noted that there was no noticeable pattern in the number of iterations to convergence with respect to K; for each of the values of K, the algorithm varied between 10 and 60 iterations with no consistency whatsoever. It is also important to note that the calculation of cosine distance is almost 4-times slower than that of Euclidean.

Running the complete suite of tests ($K = [2,40]$ with both distance metrics) took about 3 hours (with some consistency actually).

Clustering Quality and Results

With Respect to Entropy

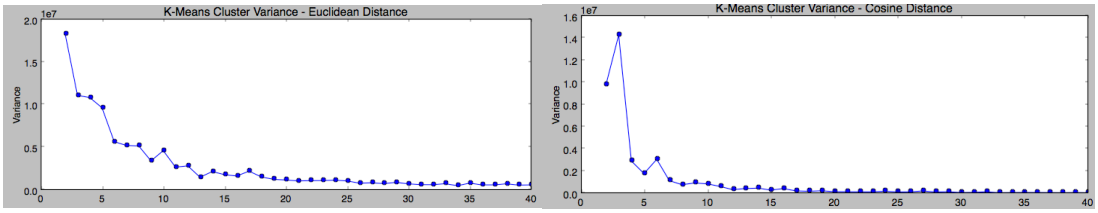
The formulation of our entropy calculation is detailed earlier in this report. For the test suite used on K-Means, the entropy of every clustering was calculated for K values varying between 2 and 40 (inclusive). The results are shown in the below graphs for both Euclidean and cosine distances respectively.



There are two big takeaways from these results: first, simply from the lack of stability in the entropy values for Euclidean distance and smaller values in the cosine distance calculations, it appears that cosine distance is a more natural metric for this dataset. One explanation for this is that documents of the same topic will tend to have several common words, but not necessarily common frequencies of those words (something that will throw off Euclidean distance). It is also important to notice that for the values of K that were tested, the optimal number of clusters with respect to entropy occurs in the neighborhood of $K = 25$.

With Respect to Variance in Cluster Size

For the most part, the variance in cluster size varied in concert with the entropy values. The below graphs show this. (Note: the values for variance were incorrectly calculated, however it is only a scaling problem, the visual pattern still holds).



As mentioned in the first paragraph, the variance in cluster sizes tends to be inversely proportional to the number of clusters K. This property follows intuitively from the way in which the variance calculation is formulated.

Other Observations

In general, the only surprising quality of this implementation was its performance. We originally expected the implementation to be very slow due to our results from DBSCAN, however, further inspection validates why this occurred. DBSCAN is and n^2 algorithm; normally this wouldn't mean much practically when compared with Kn for K-Means (only theoretically), however, in our case, the constant overhead for K-Means loses its significance when dealing with a dataset at this scale.

We performed some informal experimentation with the paired-down feature vectors from the previous lab. Naturally, the algorithm performed faster on these smaller vectors (the distance calculation was 4-times faster); but somewhat surprisingly, the results for entropy and cluster size variance remained roughly the same. There was a slight slip in these values, but not big enough to be statistically significant.

Credits

Both group members contributed equally in the implementation of this project as well as the writing of this report. Peter Jacobs implemented and tested all of the DBSCAN functionality with design input from Daniel Marchese. Daniel Marchese implemented the entropy function, as well as all of the K-Means code with design input from Peter Jacobs.