Information Retrieval 2014B - Exercise 4
Avi Gabay - 021649496
Amit Dar - 040472201

1. Design considerations:

1.1. Algorithms with same interface and interfaces in general.

While we started working on the implementation, we wanted to make sure we both provide the same functionality for all the clustering algorithms. Using basic software design, we used interfaces throughout our implementation where possible. This provided us with the possibility to create our skeleton code while the implementation of the underlying classes were not implemented. In addition, we were able to quickly find and fix classes with missing functionality.

1.2. Objects with single and focused responsibility.

Once again, in accordance with the previous section, we made an effort to make our classes as focused as possible. Once we found out a class was doing more than it should ("taking more responsibility than it should") we forked it to another class to keep it small and clear.

1.3. Using centroids instead of medoids for accurate clustering results.

Discussing what we learned in class, our choice between using medoids and centroids had to be done prior to our implementation. Keeping in mind that medoids are far more easy to handle, since we only have to calculate the item's distance from each other only once, we finally chose to use centroids. Our main goal was to provide a more accurate set of results. Using centroids helped us get closer to our goal.

2. First approach:

2.1. Implement indexing mechanism using Lucene.

Our first approach was to use Lucene as our information retrieval system by indexing documents after stripping out HTML tags and then querying on all documents thus receiving tf-idf values between all documents. This approach was later discarded; further explanation is provided in sections below.

2.2. Using Tika, then moving to JSoup.

As suggested in class, we began using Tika as our HTML stripping utility. However, because the HTML documents in our data set are very old (close to 20 years!) Tika was unable to parse such old files due to their invalid format that is already non-existent. Therefore we gave up Tika and decided to use JSoup which handled the documents without any issues.

2.3. Choosing K centroids for KMeans algorithm done by choosing every N/K document.

While implementing the basic KMeans algorithm, we had to choose our centroids seeding approach. On one hand, we wanted the seeds to mean something, on the other hand, we didn't want to use any random seeding mechanism. To establish clustering results that we can compare our future improved algorithm to, we needed to make sure we receive consistent results. Therefore we chose every N/K document as the first centroid for the KMeans algorithm. We know that while this might yield bad results on some datasets, we consider the more common input set as a spread out set, with unknown relation between the items in it.

3. **Process flow and implementation** (a general idea of how our program is running)
   - 3.1. **Parameters verification** - the program makes sure it has all the needed parameters, and that they are valid.
   - 3.2. **Document loading** - the program reads all documents from docs.txt
   - 3.3. **Document indexing** - each document is inserted into Lucene which we use as our vectorization platform. A document is analyzed using Lucene standard English analyzer and is then stemmed using Porter Stemming algorithm and inserted into Lucene.
   - 3.4. **tf-idf calculations** - each vector (document) in Lucene is read and is performed an all-to-all tf-idf calculation and is stored in a calculation matrix.
   - 3.5. **Algorithm execution** - based on the parameters from the user, the requested clustering algorithm is executed on the given data. Obviously, distances are recalculated between iterations since we are using centroids, which change their coordinates between iterations.
   - 3.6. **Results to user** - since our clustering algorithms all provide the same result interface, once we implemented a simple result handling mechanism, all our algorithms return the same result format.

4. **Performance Improvements:**

4.1. Memory usage

4.1.1. When starting out on this project, we decided to use high precision variables for our calculations in order to get improved and accurate results. Using double variables meant that our memory consumption just about doubled in size. Our calculation matrix had over 64 million cells and therefore we ran into memory consumption difficulties due to heap size exceptions (JVM consumed ~6GB RAM). Thus we rewrote our matrix data structure and used on disk memory matrix (Memory mapped file). This allowed us to save “endless” amount of results to disk. However, we ran into very slow system performance due to high I/O calls. Thus we created a third implementation to observe if using float implementation will change our results precision. Luckily, we noticed that results were completely similar and therefore changed our whole implementation to use float variables and were able to implement a full solution all in-memory decreasing memory consumption by about half and quadrupling system performance measurements (on our local machines).

4.2. Speed

4.2.1. **Implementing our own tf-idf instead of Lucene.**

As discussed in the “First Approach” section above, when building our first algorithm we decided to use Lucene queries to return tf-idf results which was then discarded due to its poor performance issues and result values. Lucene calculates tf-idf differently than the approach learned in class and therefore the result set was not as we expected. Also, performance wise, running $\frac{N*N}{2}$ (~32 million) queries was time consuming (about half a second per query) and therefore we decided to convert our usage of Lucene and make it our “vectorization” platform. After stripping the HTML document, we used Lucene tokenizers, stemmers and analyzers to produce a single vector. Based on this vector we then computed cosine similarity between vectors (documents) and centroids. Using our

own tf-idf calculation decreased retrieval time significantly (~32 million calculations in 8 seconds) and allowed us to give our own weight and boosting to vector values.

4.2.2. **Multithreaded implementation for full machine capabilities**.

After implementing the basic KMeans algorithm, we came across the most common problem, which was the long execution time for each iteration. Studying the inner process of the algorithm revealed the fact it was taking a lot of time to make all the distance calculation and attachment of each document to the closest cluster centroid. Understanding each document distance and attachment process has nothing to do with all the other documents of the same iteration, made us look for a way to use our full machine's processing capabilities (meaning - using all the available cores to calculations of the same iteration). Once we implemented this functionality, our running time decreased by an average of about 40% (on the same machine!). Considering the fact our implementation is dynamic and uses all the available cores of the host machine, running our code on a machine with more cores will yield better results.

4.2.3. **Adding finesse to clusters to avoid minor centroid movements**.

After we finished implementing all the algorithms, we started to notice a strange behavior in some of our tests. The algorithms sometime used all their available iterations (which were blocked), the results stayed the same. Digging into this issue revealed an interesting phenomena, in which the values vector of the centroid changed by a very small fraction due to two documents switching between clusters every iteration. This meant that our clusters were finalized, but two documents were constantly switching between the clusters. We understood that those documents had a very similar tf-idf vector, and after applying a mechanism of minimum distance change for the centroids (which was finally set to the value 0.0000003 in our dataset) we were able to overcome this problem.

4.2.4. **Setting maximum number of iterations for KMeans to avoid long calculations**.

In order to avoid infinite calculation loops, we added a maximum number of calculation iterations. This is done mainly to make sure we return some sort of result after a predefined number of iterations. In addition, this also helped us to reveal the problem of cluster switching documents described in the previous section (by noticing no change was made to the clusters, but the algorithm doesn't finish).

5. <u>Improved algorithm:</u>

5.1. **Enjoy the benefits of both KMeans (processing wise) and KMeans++ (quality of results wise)**.

After implementing both the basic KMeans and the KMeans++ algorithm, we started to think about the implementation of the improved algorithm. Taking into account the fact we should be basing this new algorithm on the KMeans++ algorithm, we came up with an extremely simple, yet very powerful solution, of combining the power of KMeans++ statistically random seed choices, the user input of K, and the fact we already implemented the KMeans algorithm using multi threaded approach, and basically used all those three facts to actually run the KMeans++ algorithm K times. We know it will end, since K is a finite number, and each execution of the algorithm is

bounded by a predefined maximum number of iterations. Furthermore, we now only have to make a wise choice in choosing the best results we get for each execution of KMeans++. We have two parameters (Purity and Rand Index) to test the results of each execution, and according to the importance we gave each of them, we can easily see the change of results. We are also able to give the user a way to define which parameter is more important, and base the final set of results on the user's choice.

6. Results

Algorithm	**Purity**	**Rand Index**
Basic (KMeans)	Cluster id: 1 purity: 0.401 Cluster id: 2 purity: 0.482 Cluster id: 3 purity: 0.570 Cluster id: 4 purity: 0.385 Cluster id: 5 purity: 0.646 Cluster id: 6 purity: 0.832 Cluster id: 7 purity: 0.333 Avg: 0.521	0.674
Basic++ (KMeans++)	Cluster id: 1 purity: 0.368 Cluster id: 2 purity: 0.466 Cluster id: 3 purity: 0.899 Cluster id: 4 purity: 0.561 Cluster id: 5 purity: 0.505 Cluster id: 6 purity: 0.410 Cluster id: 7 purity: 0.301 Avg: 0.501	0.666
Improve (Imp-KMeans++)	Cluster id: 1 purity: 0.589 Cluster id: 2 purity: 0.591 Cluster id: 3 purity: 0.298 Cluster id: 4 purity: 0.928 Cluster id: 5 purity: 0.597 Cluster id: 6 purity: 0.386 Cluster id: 7 purity: 0.565 Avg: 0.565	0.671