ABSTRACT
In this paper, we describe the design considerations, the implementation details and the results of performing distributed K-Means clustering on a snapshot of Wikipedia of around 13 million documents.

The design and implementation is based on the MapReduce programming paradigm. We use the MapReduce implementation provided by Apache Hadoop [1]. The running of our algorithms took place on the UC Berkeley ICluster [2].

Keywords
K-Means; Wikipedia; Clustering; MapReduce; Hadoop.

0. DISCLAIMER
Prior to writing this report, the ICluster underwent a downtime that resulted in losing the output data from our clustering. We did backup some incomplete and non-final data outside the cluster, but we haven’t had the chance to backup the latest results. Even worse, after reviving the cluster back to work, we couldn’t run any more code on it because some of our data is incomplete (some sample of the full data) and because the cluster didn’t take more jobs from users.

1. INTRODUCTION
Our dataset is a snapshot of Wikipedia that contains about 25 million documents. We describe clustering the documents in a topical way. It is worthwhile to mention that Wikipedia has its own topical clustering mechanism and the data is labeled with the Wikipedia Categories.

For each document, we construct a feature vector consisting of term frequencies in that document. The ordering of the terms is the order of the term in global counts (in descending order). We limit our feature vectors to the 10,000 most occurring words in our dictionary. Therefore, our feature vectors have length of 10,000 and sum to one.

To accomplish the clustering, we used the following pipeline. First, we performed word counts on the Wikipedia documents. Then we used the counts to tokenize the words in the documents, such that the higher the count, the lower the token id. We refer to the tokenized data as dictionary. The creation of the dictionary is discussed in section 2.

Second, we used the dictionary to construct our feature vectors (one for each document). The result is a matrix for which each row is a feature vector for a different document. The design and implementation of this step is discussed in section 3.

Then, we perform K-Means clustering in a distributed fashion on the feature vectors from the previous step. The design and implementation of the K-Means under the MapReduce paradigm is discussed in section 4.

Finally, we perform labeling to our clusters from the last step. We compare two labeling techniques. The first is using the clusters centroids as effective feature vectors and extract labels using mutual information. The second is by extracting the Wikipedia categories from the documents and matching these categories to our clusters. The labeling is discussed in section 5.

2. Creating a Dictionary
In this section, we discuss the creation of a dictionary for the Wikipedia snapshot.

2.0 WordCount
First, we perform global counting for the words in the snapshot. In our counting, we ignore HTML tags and their attributes. We tokenize the text using the Java StringTokenizer with slight additions. We also ignore punctuations when counting words. The list of ignored punctuations is presented in Table 1.

Table 1. List of ignored punctuations

<table>
<thead>
<tr>
<th>-</th>
<th>,</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>)</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&quot;</td>
<td>:</td>
<td>/</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>{</td>
<td>}</td>
<td>=</td>
<td>?</td>
<td>!</td>
</tr>
<tr>
<td>~</td>
<td>^</td>
<td>&amp;</td>
<td>*</td>
<td>+</td>
<td>;</td>
<td>‘</td>
</tr>
</tbody>
</table>

For instance, the tokens penn and teller are equivalent to the word penn&teller.

We also perform counts in a case insensitive way by transforming each line to its lower case representation prior to tokenizing and counting.

For this job, we used 500 reducers to split the outputs into separate files. The result is that the reducer yields pairs of (word, global count).

2.1 Finalizing the Dictionary
In order to build a dictionary in reverse count order (higher global counts yield lower token ids) we inverted the output from the previous step in the mapper so that the global counts become our key, and the word became our value.

Our reducer is responsible to save a global token id counter to give different ids for different words. The reducer will assign a token id to each new word it faces using the global counter, and
then increment the global counter. For this to appropriately work, we need a single counter for the whole job, and since we need the dictionary in one sorted file, we used a single reducer job. The output of this job is a triplet (token id, token global counts, word) with the token ID as the sole key of the output. The triplets represent our dictionary. The size of the dictionary is about 97 million terms.

A snippet of the highest counting tokens with their token ids is presented in Table 2.

<table>
<thead>
<tr>
<th>Token ID</th>
<th>Global Counts</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>156795873</td>
<td>the</td>
</tr>
<tr>
<td>2</td>
<td>96191637</td>
<td>of</td>
</tr>
<tr>
<td>3</td>
<td>63132126</td>
<td>and</td>
</tr>
<tr>
<td>4</td>
<td>62891272</td>
<td>in</td>
</tr>
<tr>
<td>5</td>
<td>56215206</td>
<td>to</td>
</tr>
<tr>
<td>6</td>
<td>49829092</td>
<td>a</td>
</tr>
</tbody>
</table>

### 3. The (Sparse) Matrix Representation

In this section we describe the implementation of the creation of the sparse matrix representation of the Wikipedia snapshot. The creation of the matrix was done in two stages. The first stage involves creating local counts of words, which is discussed in subsection 3.1. Then, from the local counts, we construct the feature vectors, this is discussed in subsection 3.2.

#### 3.0 Introduction

##### 3.0.1 Identifying Documents

Documents in the Wikipedia snapshot have the following format:

**Example 1: A snippet of an example Wikipedia document**

```
<page>
  <title>Example</title>
  <id>1029</id>
  <revision>
    <id>8282</id>
    ...
  </revision>
</page>
```

Where the node `page -> id` is the document ID (in this example it’s 1029). We will use this ID as an identifier of the document since it’s a unique ID.

There are two minor issues with this approach. By default, each mapper call in MapReduce will read one line from the input. This implies that when we meet the tag `<id>` in the input, it looks out of context and we need to know how decide whether it is associated it with the tag `<page>` or not. In example 1, we can see another id that is associated with the revision and not the document itself. Therefore, we made each mapper only register the first ID it meets after meeting a line containing the tag `<page>`.

The second issue with this approach is that even though each mapper will read continuously from the input, we don’t control where the division between mappers will happen. Such division may occur in the middle of a document, which means that one mapper will not meet the tag `<page>`. We decided to ignore such documents that fall between mappers. Note that the number of such documents is at most equal to the number of mappers minus one. Therefore, using ~500 mappers, the number of potential document losses can be neglected.

##### 3.0.2 Redirection Documents

It is also important to notice that there exist a lot of redirection documents in the snapshot. These documents are empty documents and just indicate redirection to another document. In our clustering approach, these documents won’t help us and, as a matter of fact, may consist noise to the K-Means algorithm. Therefore, we decided to ignore these documents and not create feature vectors for them. The final number of documents we considered for clustering is around 13 million.

##### 3.0.3 Dictionary Size

Our feature vectors consist of term frequencies in each document. Therefore, the length of each feature vector is of the same size of our considered dictionary. Recall that our dictionary without editing has around 97 million terms that obey the Power Law and more than 50% of them occur only once. We decided to use the most occurring 10,000 terms as our dictionary and construct feature vectors based on that trimmed dictionary. We also wanted to check the effect of the dictionary size (and the selection type of the terms in the dictionary, for example mutual information versus counts) on the quality of the clustering but failed to do so since the cluster fell down.

#### 3.1 Local Counts

We want to exploit the fact that all pairs of (key, value) with the same key will be analyzed by the same reducer. And since each mapper call handles one line of input and yields its own pairs of (key, value), we constructed our key itself as a pair (document ID, token ID) and let the value be the counts of that token in the specific line for a specific document. That is one reason for the name Local Counts. Our reducer takes all pairs with the same key (i.e. (document ID, token ID)) and sums their values (i.e. local counts) together gaining document counts for each token. The output of the reducer handles the document ID as its sole key and the pair (token ID, token count) as its value. Therefore resulting in sorting the output for each reducer based on the document ID only.

There’s a minor issue with this approach, which is that there’s no guarantee that the same reducer will handle all terms’ counts from the same document. The reason lies in the heart of our choice of the output key of the mapper. Recall, the mapper’s output key is a pair of (document ID, token ID), therefore, the same document ID with a different token ID is considered a different key and may be handled by a different reducer.

This issue results in having the counts of different terms from the same document potentially scattered in different output files. That is another reason for calling this output Local Counts.

Merging all the terms of the same document in the same output file was done by the next MapReduce job. This job is discussed in the following subsection. The output format of this job can be thought of as a triplet (document ID, token ID, token count) only for tokens with positive counts. These triplets are locally sorted by document ID only.
3.2 Finalizing the Matrix
We would like to have all token counts corresponding to the same document represented in a continuous fashion in the same output file. Moreover, we would like all the counts to be represented in the same line, but still in a sparse representation. This way, the K-Means MapReduce mappers will handle a full document in each call (since the whole document is represented by a single line). The mapper’s pair will have the document ID as its sole key. That’s how we will make sure the same reducer will handle all tokens from the same document. The value of the pair is itself a pair of (token ID, token count) (recall, only for tokens with positive counts).

The reducer in this job will concatenate all value pairs (token ID, token count) in a list to produce a (key, value) pair that has the following structure:

\[(\text{document ID, list(token ID, token count), (token ID, token count), ...})\]

The text representation of such a list is as follows:

\[\text{token ID} = \text{token count}, \text{token ID} = \text{token count}, \ldots\]

After running this job, we have the matrix representation of the documents where for each document we have its feature vector in a sparse representation in one line.

4. K-Means as MapReduce

4.1 Initializing the Clusters Centroids
To initialize the clusters centroids, we randomly pick data points from the Wikipedia snapshot and consider them our initial centroids. This is done by a simple MapReduce job that randomly picks data points and outputs them as cluster centroids. This is essentially the uniform sampling initialization for the K-Means algorithm.

4.2 K-Means Initial Implementation
In this section we discuss the initial implementation of the K-Means clustering algorithm under the MapReduce programming paradigm. The key trick here is that calculating the new centroids based on the previous iteration’s centroids can be done in a distributed way. Let \( K \) be the number of clusters we’re considering. In our specific implementation we considered 10 clusters and wanted to check the effect of different \( K \)'s on the clustering quality but failed to do so because of the failure in the cluster.

The input to each mapper will be the matrix representation of the Wikipedia snapshot from the previous step. In addition, all mappers will receive a file containing the clusters’ centroids from the previous iteration. The output of the reducer (one reducer) will be the updated centroids.

For each mapper, we will have two sets of cluster centroids and a counter for each cluster. The first set is simply the one from the previous iteration. The second set will contain intermediate centroid results that will be sent to the reducer to finalize. The set of intermediate results will be initialized to zero vectors at the beginning of each iteration. For each cluster, there’s a counter counting the number of documents that were closest to its centroid and were handled by the current mapper.

Each mapper will handle a subset of the Wikipedia snapshot. For each input line (recall, line = feature vector for a single document, in a sparse representation) the mapper will read the vector, find the current cluster (from the previous iteration) that is closest to it and add it to the corresponding intermediate cluster centroid (from the second centroids list). The mapper will output the pair (cluster ID, (cluster count, intermediate centroid)).

The (only) reducer will take all intermediate results from all mappers that correspond to the same cluster. Add them up and divide by the total count from all mappers. This way, we gain the new centroids in a distributed way. Refer to Figure 1 for an illustrated pseudo-algorithm for the implementation (due to [3])

**Figure 1: Illustrative algorithm for MapReduce K-Means implementation**

4.3 Improved Implementation
After running the implementation discussed in the last subsection, we observed that it could be improved by an approximate K-Means solution. Recall that each mapper handles a (somewhat random) subset of documents. The idea is that each mapper will perform multiple iterations of K-Means on its subset of documents (after saving them in memory) and combine the results only after all mappers finished doing so, in the same fashion described by the previous implementation.

Consequently, we require each mapper to save all documents’ feature vectors in memory and perform multiple update iterations on the clusters based on these points after it finished reading all of its lines (implementation detail: implemented in the Mapper.close() method). Another implementation detail is that we needed to create a MapReduce counter and keep updating it so that the Hadoop jobtracker won’t kill the task because it wasn’t reporting updates for a long time.

4.4 Epilogue
Since each job run is a one K-Means iteration in the naïve implementation or multiple K-Means iterations in the improved implementation, there is a need to chain MapReduce jobs. We performed this by writing a python script that chains these jobs simply by running them in a serial way after backing up the results from the previous iteration (on the Hadoop file system).

This improved implementation helped converge faster to the clusters centroids than the naïve implementation, as expected.

5. Cluster Labeling
After performing the K-Means clustering on the documents. We compare two ways of performing the labeling on the output clusters. The first is by using the cluster centroids as effective feature vectors, and extract the terms with highest mutual information values for each cluster. This is discussed in subsection 5.1.
The second labeling technique is done by performing maximal weight matching on a bipartite graph that consists of our clusters and the original Wikipedia topical categories. This technique is discussed in subsection 5.2.

### 5.1 Centroids as Feature Vectors

In this technique we handle the cluster centroid as if it was an actual feature vector for an actual document. Given the \( K = 10 \) cluster centroids, we can calculate 10 mutual information scores for each token (one score for each cluster). Then consider the tokens that scored highest in the mutual information test for each cluster as its labels. This was done in Matlab since it is a simple task, the results of the highest scoring labels are shown in Table 3. Please note that these were not results of the final converged data but some intermediate results since the former was lost.

#### Table 3. Top 10 labels according to mutual information

<table>
<thead>
<tr>
<th>Cluster 1</th>
<th>Cluster 2</th>
<th>Cluster 3</th>
<th>Cluster 4</th>
<th>Cluster 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>complicated</td>
<td>revert</td>
<td>sublist</td>
<td>instruction</td>
<td>babel</td>
</tr>
<tr>
<td>responded</td>
<td>history</td>
<td>disruption</td>
<td>marcus</td>
<td>babelfish</td>
</tr>
<tr>
<td>promised</td>
<td>disruption</td>
<td>votes</td>
<td>sam</td>
<td>subclass</td>
</tr>
<tr>
<td>wanderers</td>
<td>endorse</td>
<td>verifiability</td>
<td>student</td>
<td>midclass</td>
</tr>
<tr>
<td>perkins</td>
<td>hopefully</td>
<td>ignore</td>
<td>Indiana</td>
<td>preloaded</td>
</tr>
<tr>
<td>collecting</td>
<td>surprised</td>
<td>endorse</td>
<td>Pennsylvania</td>
<td>subclass</td>
</tr>
<tr>
<td>bet</td>
<td>kits</td>
<td>upser</td>
<td>William</td>
<td>lowestclass</td>
</tr>
<tr>
<td>manage</td>
<td>notify</td>
<td>laps</td>
<td>language</td>
<td>assessment</td>
</tr>
<tr>
<td>perception</td>
<td>wanings</td>
<td>touchdown</td>
<td>north</td>
<td>nominator</td>
</tr>
<tr>
<td>panthers</td>
<td>recommend</td>
<td>media</td>
<td>history</td>
<td>English</td>
</tr>
</tbody>
</table>

#### 5.2 Bipartite Graph Matching

In this subsection, we assess another way of labeling the clusters. We use two techniques described in subsections 5.1 and 5.2. The technique described in subsection 5.2 (i.e. bipartite graph maximum weights matching) lacks the data because of the failure in the cluster but is remembered to yield better labeling than the technique described in subsection 5.1 (i.e. using the centroids as effective feature vectors).

### 5.2.2 Data

With the cluster going down, we lost all the data concerning categories and couldn’t actually assess the technique described in this section, but the code for it exists and is ready to be run again when the cluster goes back up.

Although we can’t report the actual results achieved by using this technique, we recall (from before losing the data) that the labels achieved by this technique were more meaningful than the ones reported in section 5.1.

### 6. Summary

In this paper, we describe the design and implementation of clustering of Wikipedia documents given in a Wikipedia snapshot. The number of clustered documents is around 13 million documents clustered into 10 topical categories using terms frequencies in each document as a feature vector. We describe two distributed algorithms for performing K-Means. The first algorithm, described in subsection 4.2 is exact solution to the K-Means. The second improved algorithm, described in subsection 4.3, is an approximate solution but converges remarkably faster than the naive implementation of subsection 4.2.

After performing the clustering, we label the clusters using two techniques described in subsections 5.1 and 5.2. The technique described in subsection 5.2 (i.e. bipartite graph maximum weights matching) lacks the data because of the failure in the cluster but is remembered to yield better labeling than the technique described in subsection 5.1 (i.e. using the centroids as effective feature vectors).

### 7. REFERENCES


