1 Introduction

In this project, we are given a task of clustering wikipedia articles. As the data size is relatively large and cannot be memory-resident on a single node computer, we first adopt map-reduce dataflow to extract the word counts and build feature matrices. Given the compact representation of feature matrix, the clustering task is also computationally challenging due to the large number (tens of millions) of wikipedia articles. Since the category of wikipedia articles exhibits a hierarchical structure, we explore scalable clustering algorithms that are able to yield hierarchical clusters. We made our source codes available at: https://github.com/rxin/wiki-clustering.

The remainder of the report is structured as follows.

- Settings
- Feature Matrix Construction
- Scalable Clustering Algorithms

2 Settings

**EC2 Cluster.** We performed data processing on Amazon EC2, instead of the iCluster server suggested by the assignment description. Specifically, we started 10 large memory EC2 instances, which collectively have 300G of RAM.

**Spark.** Instead of using Hadoop, we chose to use Spark/Mesos cluster due to the following reasons. First, Spark provides much higher level primitives such as group-by and filter, which significantly shorten the program and provide better readability. Second, and more importantly, Spark provides an efficient distributed in-memory abstraction that is very useful in our application, since our workload exhibits strong temporal locality, i.e. we
repeatedly scan the same set of data multiple times to build our dictionary and feature matrices. We consulted Prof. Canny on Piazza and obtained his approval.

**Data Preparation.** Note that our input data is a Wikipedia XML dump file of size 37GB, which contains the latest revisions of Wikipedia articles. We first copied the XML dump onto a local folder. We then split it into 10 equally-sized pieces and uploaded them in parallel to Amazon S3, the storage nodes associated with EC2 instances. An interesting story was that our behavior caught the attention of Berkeley campus IT and they notified AMPLab admin that they saw suspicious network usage.

### 3 Feature Matrix Extraction

In this section, we report our detailed steps of processing the data and extract the feature matrices.

**Algorithm Outline.** On the higher level, our algorithm for extracting feature matrices consists of the following three steps:

1. Count the document frequency for each word, and build an encoding dictionary that maps each word to an integer index. This index is assigned in a way such that it indicates the frequency of each word, i.e., the most frequent word has an index 0.

2. Prune the encoding dictionary. We used a frequency threshold \( t \) and remove the words that appeared in less than \( t \) documents as they would hardly impact clustering performance. We experimented with different thresholds \( t \) and eventually chose \( t = 50 \). This significantly reduced the size of feature vectors.

3. Construct the sparse feature matrix. Each document ID is mapped to a hash map which itself represents a bag of words.

**Efficient Input Processing.** For efficiency considerations, we exploit various open source tools to process the relatively large scale input data in parallel. Specifically, we used `XmlInputFormat` class in Apache Mahout to split our XML dumps at the granularity of `<page>` tags. After we load the XML file into memory, we used a Wikipedia tokenizer from Apache Lucene to tokenize each page. We then passed documents through a Lucene token stream that can optionally go through either a stemming filter, or stop-word removal filter, or both. This tokenized collection of documents are then cached in memory across the cluster using Spark. Note that this process is map-only, i.e. no shuffle or reduce phase involved.

**Dictionary Construction.** To obtain the document frequency count for each word, we performed flatmap over the tokenized documents, emitting each token as a pair of `(token, 1)`. With the collection of all tokens, the frequency counting is easily achieved with
a groupByKey and reduce(_ + _). Since the sum operation is algebraic, we used a combiner on map output. This mitigates the problem of load imbalance among reducers. We used only two reducers, since the output after combiners are small from every partition. We then collected the encoding dictionary on the master node, and sent it to each downstream slaves as a broadcast variable. This is feasible because the dictionary is only 300MB in size. Note that the broadcast implementation in Spark is extremely efficient even with thousands of nodes as it uses a bit-torrent like protocol. Although we only needed ten nodes in our project, the program can scale to tens of thousands of nodes.

In Table 1, we list the first 20 entries in our dictionary, i.e., the top 20 frequent terms.

<table>
<thead>
<tr>
<th>default</th>
<th>count</th>
<th>w/o stopword</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>to</td>
<td>7587258</td>
<td>from</td>
<td>4184193</td>
</tr>
<tr>
<td>of</td>
<td>6109544</td>
<td>name</td>
<td>3816141</td>
</tr>
<tr>
<td>the</td>
<td>5715734</td>
<td>see</td>
<td>2476870</td>
</tr>
<tr>
<td>in</td>
<td>5322840</td>
<td>other</td>
<td>2312325</td>
</tr>
<tr>
<td>a</td>
<td>4719922</td>
<td>also</td>
<td>2196481</td>
</tr>
<tr>
<td>and</td>
<td>4681149</td>
<td>using</td>
<td>2157499</td>
</tr>
<tr>
<td>is</td>
<td>4297563</td>
<td>new</td>
<td>1978294</td>
</tr>
<tr>
<td>from</td>
<td>4184193</td>
<td>first</td>
<td>1788258</td>
</tr>
<tr>
<td>for</td>
<td>4029959</td>
<td>has</td>
<td>1778382</td>
</tr>
<tr>
<td>name</td>
<td>3816141</td>
<td>which</td>
<td>1771598</td>
</tr>
<tr>
<td>by</td>
<td>3761875</td>
<td>one</td>
<td>1649312</td>
</tr>
<tr>
<td>on</td>
<td>3333606</td>
<td>people</td>
<td>1620746</td>
</tr>
<tr>
<td>as</td>
<td>3268749</td>
<td>list</td>
<td>1534601</td>
</tr>
<tr>
<td>it</td>
<td>3088897</td>
<td>place</td>
<td>1504684</td>
</tr>
<tr>
<td>with</td>
<td>3057200</td>
<td>project</td>
<td>1499726</td>
</tr>
<tr>
<td>was</td>
<td>2985999</td>
<td>all</td>
<td>1485493</td>
</tr>
<tr>
<td>at</td>
<td>2702039</td>
<td>file</td>
<td>1472657</td>
</tr>
<tr>
<td>an</td>
<td>2579420</td>
<td>year</td>
<td>1471153</td>
</tr>
<tr>
<td>this</td>
<td>2504028</td>
<td>he</td>
<td>1463931</td>
</tr>
<tr>
<td>see</td>
<td>2476870</td>
<td>may</td>
<td>1412613</td>
</tr>
</tbody>
</table>

Table 1: Top 20 frequent terms

**Feature Matrix Construction.** Each slave used a partition of the cached tokenized documents and constructs a feature matrix out of all documents in this partition according to the encoding dictionary. Each feature is represented as a (key, value)-pair in the hash map, where the key is the encoded index and the value is its tf-idf value. This task is actually embarrassingly parallel, as there is absolutely no sharing or communication needed.
across the partitions. This step is also map-only. Finally, the feature matrix is written out
as hadoop (key, value) sequence files that later feed into our clustering algorithm.

The above process was repeated four times to four versions of clustering inputs, namely,
default, stemming, stop-word removal and the combination of stemming and stop-word
removal.

4 Scalable Clustering Algorithms

We now have a feature matrix representation of the data. In this section, we discuss different
hierarchical algorithms and how they can be applied to our task.

**Bottom-up Approach.** For each wikipedia page, we have a feature vector of tf-idf
values of all the terms. We normalize each vector by $L_2$-norm. To cluster the documents,
we first implemented a hierarchical agglomerative clustering, where the similarity of two
clusters is defined as the similarity of their centroids

$$SIM(C_i, C_j) = (\frac{1}{|C_i|} \sum_{d_m \in C_i} d_m)(\frac{1}{|C_j|} \sum_{d_n \in C_j} d_n)$$

where $d_m$ is the unit vector representation of the document $m$. Our implementation follows
the pseudo-code given in the book [?], Chapter 17.

We tested our implementation on a relatively small sample, and the clusters we get have
reasonably larger in-cluster similarity comparing to the average similarity of the entire
dataset. We realized, however, that this algorithm was not scalable to the data size of
our task. Even with some efforts to heuristically avoid computing similarities for a lot
of document pairs, we still need to compute the similarity between $O(n^2)$ dot products
in order to get reasonable clusters. Given that $n \approx 10^7$ in our case, we estimate that our
program needs to run several weeks to just get some initial pairwise similarity scores.

**Recursive Bisection.** We also implemented a much more efficient hierarchical clustering
algorithm based on recursive bisection. The idea, described in [?], is as follows. To bisect a
set of documents, two documents are randomly picked upfront as seeds. For each document,
we compute its similarities against the two seeds, and associate this document with one
of the two seeds that has a higher similarity score. For better performance, we repeat the
above process ten times and each time we randomly chose seed documents independently.
Finally, we take the bisection that gives the best quality. The quality of a cluster we used
is defined as the sum of the cosine similarities of each document in the cluster with the
centroid of the cluster, i.e., $D_C = \sum_{d_i \in C} d_i$. It is easy to see that the quality measure of a
cluster $C$ is simply the $L^2$-norm of $D_C$.  

4 of 5
We start with all the documents in a single cluster, and recursively bisect the cluster. At each iteration, we always choose the bisection that leads to the best improvement of the quality measure. Specifically, we tentatively bisect each cluster $C$ into $C_1$ and $C_2$, and carry out the bisection on $C$ which maximizes $|D_{C_1}| + |D_{C_2}| - |D_C|$.

This algorithm runs much faster. Each bisection step takes time linear to the number of documents in the cluster, and if the result is not too unbalanced, the total running time is linear to $n$.

Due to time constraints, we have not finished running the cluster algorithm over the entire feature matrix. But according to [?], the algorithm is scalable to our data size and generally produces good clustering results on text data.