Introduction:

Clustering is an important machine learning task that tackles the problem of classifying data into distinct groups based on their features. An ideal clustering algorithm maximizes feature similarities within a cluster while minimizing the feature similarities across clusters. Some of the most common clustering algorithms include spectral clustering and k-means clustering. This project essentially consists of two main parts: extracting features into a bag-of-words representation and then performing clustering using these features.

In this assignment, we are given a 37 GB snapshot of Wikipedia stored in XML format. Our goal is to cluster the Wikipedia articles into hierarchical categories based on their features. In this case, the features consist of word occurrences within each article. We first construct a bag-of-features representation of the data. To begin, we must assign each word a unique ID where the IDs are given to words based on the number of occurrences of each word. Once we create our dictionary, we can transform each of our articles to a bag-of-words representation. Next, with our bag-of-words we perform the clustering using the spectral clustering algorithm to seed k-means.

Setup Configuration:

We performed our clustering using a somewhat different setup from the one specified in the assignment (although ours was also approved by Prof. Canny). Instead of running Hadoop on the iCluster, we used Spark on Amazon EC2 to create our dictionary and build the feature matrix. Our reasons for this choice were that in addition to supporting Hadoop’s MapReduce paradigm, Spark provides a number of other high-level operations that simplified our implementations considerably. Moreover, Spark’s support for distributed in-memory caching of data sets significantly improved performance by allowing us to keep the dictionary and frequency count data cached in memory, which dramatically reduced the I/O overhead that would have been involved in the feature extraction stage, had we chosen to use Hadoop.

We uploaded the 37 GB Wikipedia XML dump file to Amazon S3 so that it could be used by EC2 instances. We then started a 10-node EC2 cluster consisting of “Large” instances with 7.5 GB each, for a total of 75 GB. We used the Mesos EC2 setup scripts to do this, which automatically set up both Spark and HDFS on the cluster nodes. We then transferred the XML data into HDFS, which automatically partitions and distributes the file across the local disks of each node.

Once the data was in place on the cluster, we used Spark to process the data into a dictionary and feature matrix. Finally, we began to use BIDMat to cluster the results generated from Spark.
Feature Matrix:

On a high-level, creating the feature matrix requires first parsing and tokenizing the XML data, creating a dictionary that maps IDs to words, and then converting each article into its bag-of-words representation using this dictionary.

Below, will we describe the major steps in depth. We also provide a walk-through of some of our code, to underscore both the algorithms we used as well as to highlight the powerful flexibility of Spark’s high-level operators vis-a-vis Hadoop’s more limited map and reduce. We performed most of our Spark jobs in an ad-hoc manner using the spark-shell, so we have provided the scala code here.

1) The first thing we must be able to do is tokenize the input data so that we can extract the relevant words from each article. We used the `XMLInputFormat` provided by Apache Mahout to enable us to read records from the dataset based on the `<page>` and `</page>` tags. This worked well for us and allowed us to read in a single article at a time.

Next, we needed to extract the relevant tokens from each article. We used the Wikipedia tokenizer from Apache Lucene. Once we were able to parse the article data, we needed to create our word dictionary. This involves outputting the key-value pair (word,1) for each article. After the map-step, we use multiple reducers to merge the counts for each word. In spark, this process basically translates to:

```scala
val rdd = SparkContext.hadoopFile("wikipedia.xml")
val wordCounts = rdd.flatMap(article => extractTokens(article))
                .reduceByKey(_ + _)
```

Spark automatically uses combiners, so the process was quite easy for us. The next step involves computing a total ordering of the words based on their counts in descending order. We map each (word, count) pair to (count, word) and then we want to sort by counts. Spark has a distributed `sortByKey` function (implemented by Antonio!) that uses a range-partitioner to partition the data by key so that it can sort the data on multiple reducers simultaneously. In Spark code, these steps translate to:

```scala
val ascending = false
val sortedCounts = wordCounts
                  .map((word, count) => (count, value))
                  .sortByKey(ascending).collect()
```

This collects the words sorted by count in descending order. We can assign IDs to the words by doing:
Table 1: Top 10 Words

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>to</td>
<td>7587258</td>
</tr>
<tr>
<td>of</td>
<td>6109544</td>
</tr>
<tr>
<td>the</td>
<td>5715734</td>
</tr>
<tr>
<td>in</td>
<td>5322840</td>
</tr>
<tr>
<td>a</td>
<td>4719922</td>
</tr>
<tr>
<td>and</td>
<td>4681149</td>
</tr>
<tr>
<td>is</td>
<td>4297563</td>
</tr>
<tr>
<td>from</td>
<td>4184193</td>
</tr>
<tr>
<td>for</td>
<td>4029959</td>
</tr>
<tr>
<td>name</td>
<td>3816141</td>
</tr>
</tbody>
</table>

2) Next, we decided to prune the dictionary to make further computation more manageable, leaving in the top 50,000 most frequently-occurring words. Once we had a pruned dictionary of the most frequent words, we also wanted to experiment with stemming to see how that would affect clustering. Stemming is used to mitigate the effects of varying suffixes and inflected forms of common morphological roots in words. We also considered experimenting with other feature representations including bigrams and tf-idf.

To do this, we used ScalaNLP’s built-in Porter Stemmer implementation to derive the root of each dictionary term. We then built a new dictionary of unique roots, mapping them to the sum of the frequencies associated with the corresponding terms in the unstemmed dictionary. Finally, we sorted this list of counts (as described above) and took the first 50,000 entries. We performed all of this computation on the master node since the dictionary is fairly small. Within the spark-shell, we used the following scala statements:

```scala
def SortedCount = sortedCounts.map((count, word) => word).toArray

Table 1: Top 10 Words

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>to</td>
<td>7587258</td>
</tr>
<tr>
<td>of</td>
<td>6109544</td>
</tr>
<tr>
<td>the</td>
<td>5715734</td>
</tr>
<tr>
<td>in</td>
<td>5322840</td>
</tr>
<tr>
<td>a</td>
<td>4719922</td>
</tr>
<tr>
<td>and</td>
<td>4681149</td>
</tr>
<tr>
<td>is</td>
<td>4297563</td>
</tr>
<tr>
<td>from</td>
<td>4184193</td>
</tr>
<tr>
<td>for</td>
<td>4029959</td>
</tr>
<tr>
<td>name</td>
<td>3816141</td>
</tr>
</tbody>
</table>
```
val stemmedDictionary = stemmedUngrouped
                .groupBy(_._1)
                .map((word, counts) => counts
                    .reduce(_ + _))
                .sortBy(_._2)
                .map(_._1).toArray

3) To construct the sparse feature matrix, we mapped each document ID to an array of
tuples containing the word ID and number of occurrences of each term contained in that
article.

val encoder = new HashMap[String, Int]()
stemmedDictionary.view.zipWithIndex foreach {
    case (term, index) =>
        encoder += term -> index
}

val featureMatrix = rdd.map(article => extractTokens(article))
                    .map(porter _).map(encoder.get _)
                    .filter(stemmedDictionary.contains _)
                    .toSeq.groupBy(identity)
                    .mapValues(_.size)

Note that we could have cached the RDD containing the extracted tokens for each
article in the first step by doing the following:

val tokensRDD = rdd.map(article => extractTokens(article))
tokensRDD.cache()

This would have loaded the RDD containing the tokens into our cluster’s memory,
eliminating the overhead of having to re-extract the tokens to create the feature matrix.
This way, we would only need to do the following:

val featureMatrix = tokensRDD.map(porter _) // and so on, as
above, avoiding token extraction

Clustering:

After we have created the feature matrix, the only step left is to actually perform the
clustering. The two main clustering algorithms that we considered were k-means
and spectral clustering. We planned to use the BIDMat library to perform the matrix
operations for clustering, but unfortunately ran out of time and weren’t able to finish
getting the results. Nevertheless, we will briefly describe the algorithm we experimented
We began to implement a spectral clustering algorithm to generate the first set of cluster points, then apply the standard k-means algorithm to these points. The main steps of this algorithm are as follows:

For a weighted graph $G = (V, E)$, with the set of vertices $V = \{v_1, v_2, \ldots, v_n\}$, we want to cluster the vertices into $k$ clusters. $E$ is the set of edges such that each $e_{ij} \in E$ is associated with a weight $m_{ij}$. In our case, we construct the graph with edges between words the co-occurred in articles. The weights ($m_{ij}$) would be computed based on the frequency of co-occurrence of these words in the documents. Spectral clustering works by bounding the conductance of the graph.

First, we form an affinity matrix $A$:

$$A : V \in \mathbb{R}^{n \times n} \quad \text{where} \quad A_{ij} = m_{ij} \text{ if } i \neq j \text{ and } A_{ii} = 0$$

Next, we create the diagonal matrix $D$:

$$D_{ij} = \sum_j A_{ij}$$

Then we construct the matrix

$$L = D^{-1/2} A D^{-1/2}$$

We find $x_1, x_2, \ldots, x_k$, the $k$ largest eigenvectors of $L$ and the matrix $X = [x_1 \ x_2 \ \ldots \ x_k] \in \mathbb{R}^{n \times k}$

Then we normalize $X$ to create the new matrix $Y$:

$$Y_{ij} = X_{ij} / (\sum_j X_{ij}^2)^{1/2}$$

Finally, we would treat each row of $Y$ as a point in $\mathbb{R}^k$ and apply the standard k-means algorithm on these seeds to cluster the points into $k$ clusters.