CS294_1 : Assignment 3

Natarajan Chakrapani, Kuldeep Kapade, Gaurav Shetti

We broke up the assignment into 2 parts
- creation of the sparse matrix via Hadoop
- Running a k-means clustering algorithm devised in MATLAB to determine the categories.

PART 1
Creation of Sparse matrix via hadoop

- As indicated in the assignment spec, we went about analyzing the Wikipedia xml and devising a set of hadoop jobs to form a final sparse matrix. This was broken into a set of hadoop jobs namely,
- Job 1 : Take input Wikipedia.xml and retrieve the content incorporated in the <text> tag for each <page>. We used the jdom parser in our hadoop job to iterate through the DOM tree considering content between all <page> and </page> tags. We operated on a small sample xml extracted from the original Wikipedia.xml
- Code reference : Job 1 : job1/ParserDriver.java ; job1/MyParserMapper1.java and job1/InputFormatXml.java
- Approach : We took 400 lines of Wikipedia.xml text which comprised three <page> elements to test our code on. We also observed that there were a lot of redirect pages with no content in them. These pages has #REDIRECT keyword specified in the <text> tag in the page, and we aimed to eliminate them as they added to the noise. Our mapper function parsed the xml via jdom parser and emitted the token as the value with a null key. there was no reducer needed for our purposes here. We also checked for case sensitivity of tokens by making sure all our tokens were of the same case and made sure that many contiguous whitespaces were merged into one, so as not
- Output : we got all the tokens contained in each of the pages into the mapper output location

- Job 2 : Using the token output from Job 1, we ran the WordCount java program on it to get a token frequency count of all tokens across the text corpus. Mapper function comprised of parsing the input on whitespace and emitting a (word,1) event for the reducer to merge the counts.
- The no.of unique tokens we discovered was 27425656 (~ 27.4 million)
- Code reference : job2/WordCount_original.java

- Job 3 : the output from job 2 was not sorted by the word frequency count, which we needed to create the dictionary. Hence, we tweaked the WordCount program, and gave the input to the mapper with count as key and value as the token and allow the reducer to emit the (frequency,token)
tuple as the output via a SINGLE reducer, which was sorted by the key values in ascending order.
- Code Reference: job3/WordCount.java

- To reverse sort our dictionary, I loaded the output file from job 3 onto a vi editor and reversed the entire file to ensure the highest frequency word was listed at the top line. The line number was also added, as a token id, to each line to correspond to the token. We now eliminated tokens from this set, which had any digits in them, or comprised of punctuation characters like _. We narrowed our dictionary further to 80000 unique tokens by eliminating all tokens with a frequency count less than 5.

Sample commands used:

```bash
cat /scratch/10002.txt | awk '{if($2!=3)print;}' > /scratch/10003.txt
cat /scratch/10003.txt | awk '{if($3!~/[0-9]/)print;}' > /scratch/10004.txt
cat /scratch/10004.txt | awk '{if($3!~/_/)print;}' > /scratch/10005.txt
cat /scratch/10005.txt | awk '{if($3!~/^[a-zA-Z]/)print;}' > /scratch/10006.txt
```

Task 2: creation of hash map from the dictionary.
Using the reverse sorted dictionary, we proceeded to create a hashmap with the key being the token and the token_id being the value. We proceeded to save that into a serialized file.
Code Reference: task2/FileRead.java

Task 3: Here we used the serialized file and created a hadoop job, using similar code from job 1, to read in the original Wikipedia.xml to parse the <text> tokens and lookup the token with its corresponding token_id from the hash map. Each mapper, opened the serialized file and read it into a internal Hash map object, allowing the lookup to happen in parallel.

We skipped the parsed token, in case if we did not find it in the dictionary.

We also noted the <id> in each <page> segment to correspond to the document we were parsing, to get a document id to create a tuple like (document id, token id) as an output from our reducer code.

To create the sparse matrix, we appended a int value of 1 to the existing tuple of (document id, token id) to make it into a 3 member tuple of (document id token id, 1) using the unix awk command.
At the end we obtained a (document id, token id, 1) tuples which we used via MATLAB to convert to a sparse matrix using the spconvert() function.

- Code reference: task3/ParserDriver.java; task3/MyParserMapper1.java and task3/InputFormatXml.java
PART 2: **K-means**
We had designed the Kmeans code in MATLAB.

Code reference: kmeans.txt

Input: The feature words or the feature matrix computation from hadoop be accepted as input
Output: The clustered data

Analysis:
We chose Matlab to process the computed matrix since we were familiar with the sparse operations in Matlab. In Kmeans we needed initial means or categories.

Choice of mean:
The way we had decided to choose the mean was to select those documents which had only 1 category assigned to it. In this way, when the documents would get clustered into the mean document, it would have just 1 category assigned to it rather than having multiple categories assigned. Initially since we had seen 12 well known categories, we would choose those documents which had these 12 categories assigned to them uniquely, 1 document for each category. If there are more than 2 documents having just 1 category assigned to it, we would resolve it by choosing the document with highest number of unique tokens. This was done with the help of a unix command (since the sparse matrix had the document id and the feature id).

Kmeans:

- We would then compute the euclidian distance $d$ of each document from each of the means. (Each feature would be a dimension and in such an n dimensional model, each document would represent a particular vector). The one with the least distance would be recorded as +1 in that cluster and the other value of the document with respect to other clusters would be set to 0. Thus in the end, we would get the actual document categorization.
- After the documents are categorized ; their means would be recomputed and the entire process would be repeated again to recluster the documents. We had initially chosen the threshold iterations to be 10 to check if the documents would converge after these iterations. But since we really couldn’t test this, we were ready to do trial and error upto 25 iterations.
- The final converged documents would then have the category of the mean assigned to it.

If in the above process, if we would have a category or a mean under which no documents converged, we would remove that category/document/mean from the above process and reiterate the entire process from scratch.