LAB4: LARGE SCALE DATA (TEXT) PROCESSING WITH HADOOP MAPREDUCE

1. What's trending?: Wordcount on tweets

Collect tweets using the approach you used in Lab1. However this time you will collect tweets about a certain domain, say, soccer or economy. Run "wordcount" on the tweets (may be on the @word and #tags) and visualize the output using "tag cloud" or "word cloud'. (This can be dynamic and realtime too!) Use the VM you installed in the preparation for this lab.

Input: Tweets for a given domain
Output: Word-cloud for the Input
Processing: MR on HDFS

Solution:

1. The initial tweet collection and cleaning i.e. removing retweets and getting only the text has been done using R in Jupyter notebook. ⟶ **Lab4_Activity 1.ipynb**
2. Word Count code using MapReduce has been written in JAVA. ⟶ **Activity1.java**
3. I tried to install wordcloud "R" package in numerous ways but was unable to install it, does I have made the wordcloud with two different approaches.
    a. Used online site - http://www.wordclouds.com/ to generate the wordcloud from the frequency matrix obtained from the MapReduce.
    b. Making a text (**text.txt**) file using frequency matrix in python (**wordcloud.py**) obtained from MapReduce and passing it through http://tagcrowd.com/
4. Both the images of wordcloud can be found in ancillary folder.

2. Word co-occurrence on tweets

First step in sentiment analysis is the co-occurrence of the topic of interests with words representing good or bad sentiments. We will not perform sentiment analysis. We will do just word co-occurrence. Perform word co-occurrence as described in Lin and Dyer's text [3], with pairs and stripes methods for the tweets collected for the step above. Of course, use MapReduce approach with the data stored on the HDFS of the VM you installed.

Input: Tweets
Output: Co-occurrence pairs and stripes
Processing: MR on HDFS

Pairs:

1. Made a WordPair class which supports basic structure of key and its neighbour.
2. For every line Mapper makes a Key- Neighbour pair and emits count one for each pair.
3. The combiner is same as the Reducer.

4. Reducer takes the all the same combinations together and emits the number of such pairs present in the document.
5. We consider an entire line as the window while making pairs.

    Input of Reducer : <Text, IntWritable>
    Output of Reducer : <WordPair, IntWritable>
    Input of Reducer : <WordPair, IntWritable>
    Output of Reducer : <WordPair, IntWritable>

Stripes:

1. Made a new map structure - myMapWritable.
2. For every line Mapper makes a Key- Neighbour pair and emits count one for each pair.
3. The combiner is same as the Reducer.
4. Reducer takes the all the same combinations together and emits the number of such pairs present in the document.
5. We consider an entire line as the window while making stripes.

    Input of Reducer : <Text, myMapWritable>
    Output of Reducer : <Text, myMapWritable>
    Input of Reducer : <Text, myMapWritable>
    Output of Reducer : <Text, myMapWritable>

3. Featured Activity 1: Wordcount on Classical Latin text

This problem was provided by researchers in the Classics department at UB. They have provided two classical texts and a lemmatization file to convert words from one form to a standard or normal form. In this case you will use several passes through the documents. The documents needed for this process are available in in the UBbox [7].
Pass 1: Lemmetization using the lemmas.csv file
Pass 2: Identify the words in the texts by <word <docid, [chapter#, line#]> for two documents.
Pass 3: Repeat this for multiple documents.

Output Format:

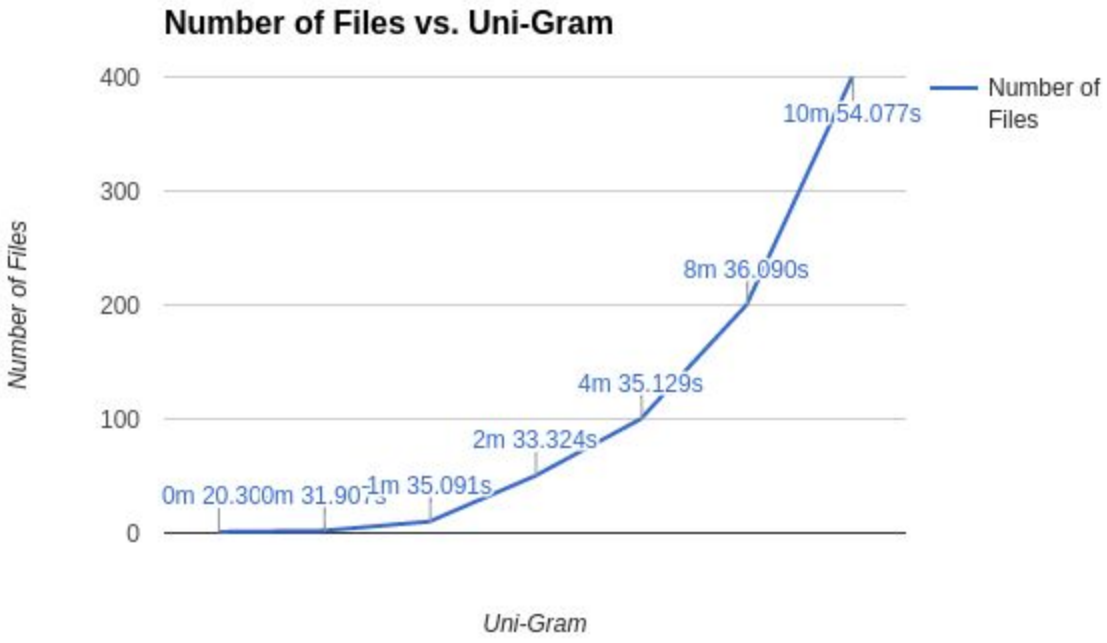
Word <loc1> <loc2> <loc3>... <count = n>
Lemma1 <loc1> <loc2> <loc3>... <count = n>
Lemma2 <loc1> <loc2> <loc3>... <count = n>
Lemma3 <loc1> <loc2> <loc3>... <count = n>

Process:

1. Made a separate class for Lemmatization. This is used to one time make a hashmap for the lemma file which has be to used again and again by the Reducer.

2. Mapper reads the input files and separates out the location and the text to be tokenized. Punctuations are removed as well as the text is lowercased for better performance.
3. Reducer takes in the different tokens and clubs them to emit a word and its location.
4. Then is any lemmas exists for the word then it emits the lemmas with the same location as the word.

Performance Graph:



4. Featured Activity 2: Word co-occurrence among multiple documents.

a. In this activity you are required to "scale up" the word co-occurrence by increasing the number of documents processed from 2 to n. Record the performance of the MR infrastructure and plot it as discussed in Chapter 3 of Lin and Dyer's text[3]. Also see the performance evaluation charts on p.56 (60) of the same text.

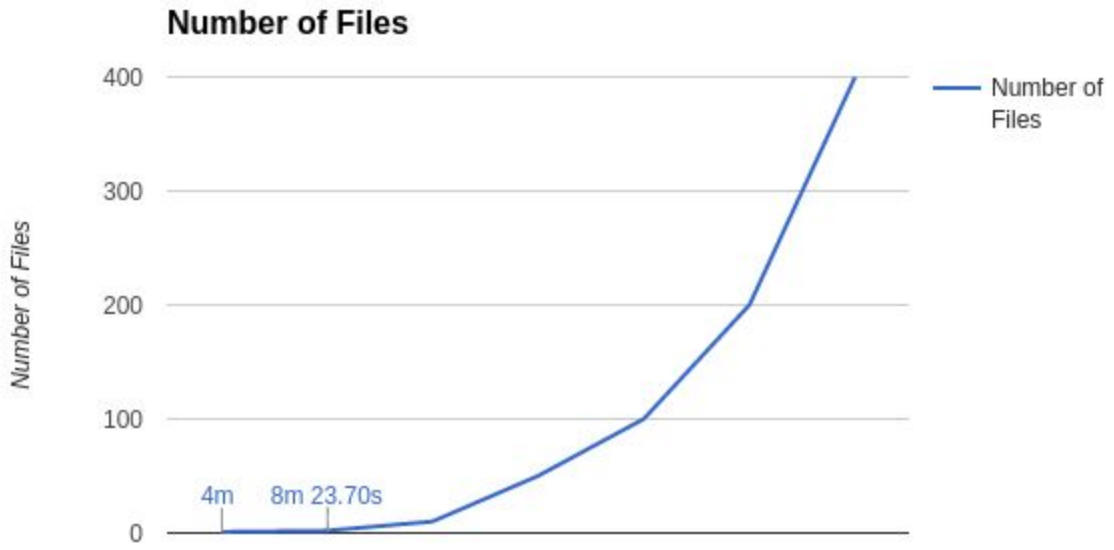
Output Format:

<word = word> <neighbour = next word> <loc id> <count>
<word = lemma1> <neighbour = next word> <loc id> <count>
<word = lemma2> <neighbour = next word> <loc id> <count>
<word = word> <neighbour = lemma1> <loc id> <count>

Process:

1. Used pairs approach to solve the problem.
2. Added separate class for Lemmatization, so that its done only once.
3. Lemmatization is done in Reducer.

I believe there is a tradeoff between correctness and performance here, It'll be better if add lemmatization process to mapper but its slows the code a alot.



For files greater than 2 the code times out due to memory full error.

b. From word co-occurrence that deals with just 2-grams (or two words co-occurring) increase the co-occurrence to n=3 or 3 words co-occurring. Discuss the results and plot the performance and scalability.

Output Format:

<word = word> <neighbour = next word> <next neighbour = next to next word> <loc id> <count>

<word = lemma1> <neighbour = next word> <neighbour = next word> <loc id> <count>

<word = lemma2> <neighbour = next word> <neighbour = next word> <loc id> <count>

<word = word> <neighbour = lemma1> <neighbour = next word> <loc id> <count>

Process:

1. Modified the pairs approach to add next neighbour as well.
2. Added separate class for Lemmatization, so that its done only once.
3. Lemmatization is done in Reducer

For 1 file the runtime is: 1 hour 10 mins.

Thus as the n-grams increases the time taken to process increases exponentially.