Big Data Analytics

Assignment 2

Arjun Subramanyam Varalakshmi (1546081)

Problem Description:

In this assignment, we will be dealing with two of the classic Natural Language Processing (NLP) problem namely TF-IDF and Word2Vec using a cluster computing framework known as Apache Spark.

TF-IDF (Term Frequency – Inverse Document Frequency) is a metric used to determine the importance of a word to the document in a large collection of documents. The term. It's combination of two numerical statistics, first statistic being term frequency which measures how many times a word occurs in that document, as we treat all words equally while computing TF it allows some of the stop words to have high frequency, which are of no importance, hence to tackle this problem we use IDF which diminishes the weight of terms which are very common in the documents and increase the weightage of words which are important. We use Spark MLlib to generate a TF-IDF model, HashingTF is used for TF and IDF is used to compute IDF. In this assignment, we calculate the TF-IDF for unigrams and bigrams generated for large collection of documents. We use Spark MLlib NGram function to computer unigrams and bigrams.

Word2Vec is a class of models which represents individual words as vectors in the feature space. This model is based on the co-occurrence statistics amongst the words in a collection. Once the vectors are computed for the words, we can use this model for determining the similarity between two terms (words), by computing the distance between the vectors in the feature space. We will be using Word2Vec algorithm in Spark MLlib, it uses skip-gram model which learns vector representation by considering the context in which words occur.

The assignment consists of 3 parts:

Part – 1: Computing term frequency without MLlib and analyze the impact of change in number of executors (2, 5, 10) on the performance

Part – 2: Computing TF-IDF model using MLlib for unigrams and bigrams generated on collection of books and analyze the impact of change in number of executors (5, 10, 15) on the performance

Part – 3: Computing Word2Vec model for large collection of books using Spark MLlib, analyze the impact of change in number of executors (5, 10, 15) on the performance and determine the synonyms for 10 words using the word2vec model generated.

In this assignment, we would be using Apache Spark along with Hadoop Framework (HDFS, YARN) to complete the assignment.

Solution Strategy:

We would be solving the above problem using Apache Spark cluster computing framework and its advanced data structures such as RDD (Resilient Distributed Dataset) and DataFrames to solve the above problems.

The reason for choosing Apache Spark over other cluster computing frameworks is its ability to provide maximum utilization of resource without compromising on key features such as fault tolerance, scalability and high performance. It also uses facilitates python API using PySpark framework, which provides simple python interface for writing the code and provides various configuration options to optimize the execution.
There are different possible solutions which can be provided using the PySpark, the approach followed by me to solve this problem is as described below:

Step 1:

Input:

HDFS directory containing list of books:

/cosc6339_s17/books-longlist

Output:

Files containing Document name, word and it's count in the document:

/bigd21/hw2_step1_output/part*

Key-Steps:

Using WholeTextFiles method read all the files in the input directory which produces RDD containing <K, V> pair of <filename, Contents of file>.

Perform transformations on the RDD to produce a RDD of the format (<Document, Word>,1)

Perform reduceByKey transformation to obtains count per book

Finally perform SaveAsTextFile action to store results onto text file

Note: For the Part-1 data gets loaded only when you are performing SaveAsTextfile, till then we just define what needs to be done on RDD, before an action is performed.

Step 2:

Input:

HDFS directory containing list of books:

/cosc6339_s17/books-longlist

Output:

Files containing unigrams and their TF-IDF sparse vector

/bigd21/hw2_step2_unigram_output/part*

Files containing bigrams and their TF-IDF sparse vector

/bigd21/hw2_step2_bigram_output/part*

Key-Steps:

Using WholeTextFiles method read all the files in the input directory which produces RDD containing <K, V> pair of <filename, Contents of file>.

Using Tokenizer split the contents of the file into a wordlist which is converted to DataFrame and used by unigram_tfidf and bigram_tfidf methods

NGram function from Spark machine learning library is used to generate unigrams by passing value of n as 1 and generate bigrams

We use HashingTF algorithm from Spark MLlib to generate the TF value for the word collection, by default it supports 2^20 words without collision, we can increase this value by passing the new value as parameter to HashingTF object

We Use IDF algorithm from Spark MLlib to generate the IDF value for the word collection and transforming the dataframe generated by hashingTF generates TF-IDF sparse vectors containing TF-IDF value for each word.

The sample output of sparse vector is (261420 {13:0.9877……})

where first value represents the size of vocabulary in the collection and the next <K, V> pair represents the hash value of word in the doc and it's TF-IDF value.

HashingTF is unidirectional we can't get back word using hash value generated by the algorithm.

Step 3:

Input:

HDFS directory containing list of books:

/cosc6339_s17/books-longlist

Output:

Files containing Word2Vec model

/bigd21/step3_word2vec_model/part*

Files containing top 10 words

/bigd21/synonyms_input/part*

Files contraining synonyms for words

/bigd21/synonyms/<directory with wordname>/part*

Key-Steps:

Using WholeTextFiles method read all the files in the input directory which produces RDD containing <K, V> pair of <filename, Contents of file>.

Create a DataFrame from the RDD after tokenizing the content into words and pass this data frame to fit into Word2Vec object created using Word2Vec class from the SparkMLlib.

I have computed TF-IDF for the collection using mathematical formula of TF-IDF and by ordering them based on TF-IDF choosing the top 10 words from the corpus to generate the synonyms.

By Passing the word and the number of synonyms required to the model generated using word2vec by passing the tokenized word collection, we can get synonyms for the words.

Word2Vec model uses skip-grams model with soft-max which generates the synonyms according to their context instead of just considering their plain English meaning.

Alternate Solution: Can Randomly pick the 10 words by sampling the vocabulary using sample function.

Why I didn't use alternate solution?

Since we are interested in learning the TF-IDF model and its significance in the text analytics I chose this approach to choose the words rather than random sampling of words.

Code Explanation:

Step 1:

- We would provide a directory path consisting of list of books as input to the program.
- WholeTextFiles is used to read input and produces an RDD consisting of filename and it's whole contents
- We would use flatMap transformation on the generated RDD and would use processFile function to return a key value pair of the form (<doc, word>,1) and returns it
- We would use reduceByKey on the returned values by process File function since key is <doc,word> which identifies the word belonging to doc. We would get word count per doc.
- We use saveAsTextFile to save the output to the textfile , all the operations defined till now using transformations will actually occur now, since only when action is performed the data actually loads into memory.

Step 2:

- In the Second part, we would be computing TF-IDF model for the given collection of books.
- We would provide a directory path consisting of list of books as input to the program.
- WholeTextFiles is used to read input and produces an RDD consisting of filename and it's whole contents.

- Use the map transformation on RDD and remove the special characters and numbers from the text
- Use the tokenizer to tokenize the text and convert it to DataFrame, which would be used as input to NGram model to generate unigrams and bigrams
- We would pass the generated unigrams and bigrams data frames to the HashingTF object by mentioning the input col and output col
- The computed result of hashing TF is passed to IDF to generate TF-IDF model for the collection of books.
- The output for unigrams and bigrams written separately to two separate directories.

Step 3:

- In the third part, we would be computing Word2Vec model for the given collection of books, which can be used to represents individual word as vectors in the feature space. This model is based on the co-occurrence statistics amongst the words in a collection. Once the vectors are computed for the words, we can use this model for determining the similarity between two terms (words).
- We would provide a directory path consisting of list of books as input to the program.
- WholeTextFiles is used to read input and produces an RDD consisting of filename and it's whole contents.
- Use the map transformation on RDD and remove the special characters and numbers from the text
- The generated DataFrame is passed to word2vec object to generate the word2vec which would create vector representation of word in the feature space.
- Word2Vec model can be used to find the synonyms for the given word, the only the condition the word given as input to find synonyms must be part of the vocabulary.
- 10 words have to be chosen to get the synonyms, here I have written a method named getWordsToFindSynonyms which will create a file containing top 10 words in the corpus based on TF-IDF value.
- This method is commented in the code submitted and the top 10 words are hard code as list, in order to verify the top 10 words can un comment the function call and run it to get the top 10 words.
- Using loop we would find the synonyms for the word and save the synonyms for that word in a path /bigd21/synonyms/<directory with word name>

How to run the code:

1. Unzip the HW2_1546081.zip folder and place the contents of the folder in the whale cluster.
2. cd to HW2 directory, it consists of a shell script named run_hw2.sh, to run the shell script give below command in terminal:
    a. ./run_hw2.sh

b. If in case it gives permission error change permission using below command and execute it
    i. chmod +x run_hw2.sh (or chmod 777 run_hw2.sh) and execute.
3. The shell script would provide you with following 4 options:
    a. Run step1, step2, step3, run all and quit.
    b. If first 4 options are selected the script asks the user to enter the number of executors he would like to use to execute the code with.
4. If in case it's required to change input directory path or how the input is given you need to edit the shell script and execute.

Results:

Description of Resources used:

The Assignment – 1 of this course is executed on Whale cluster, The components of Whale cluster can be as described below:

It consists of 57 Appro 1522H nodes

Hardware Details:

- CPU: Two 2.2 Ghz quad core AMD Opteron processor (8cores total)
- RAM: 16 GB
- Ethernet NIC: Gigabit Ethernet
- 4xDDR InfiniBand HCAs

Network interconnect:

- 144 port 4xinfiniBand DDR Voltaire Grid Director ISR 2012 switch (donation from TOTAL, shared with crill)
- two 48 port HP GE switch

Storage:

- 4 TB NFS /home file system
- 7 TB HDFS file system (using triple replication)

Software Details:

- Hadoop Framework version 2.7.2 (distributed – mode)
- Apache Spark 2.0.2 (Spark MLlib, Spark SQL, Spark Core)
- PySpark Framework
- Python version 2.7.2

- Bash scripting

Description of measurements performed:

For every part, we would analyze the impact of change in the number of executors on the performance of the code. The analysis for each part is shown below

Part -1

No. of Executors	2	5	10
Avg Time Taken (5 readings) (in seconds)	59	52	43
Min Time Taken	54	50	41
Max Time Taken	83	63	65
Time Taken By PYDOOP	72	70	68

Table 1: Time taken by Part - 1

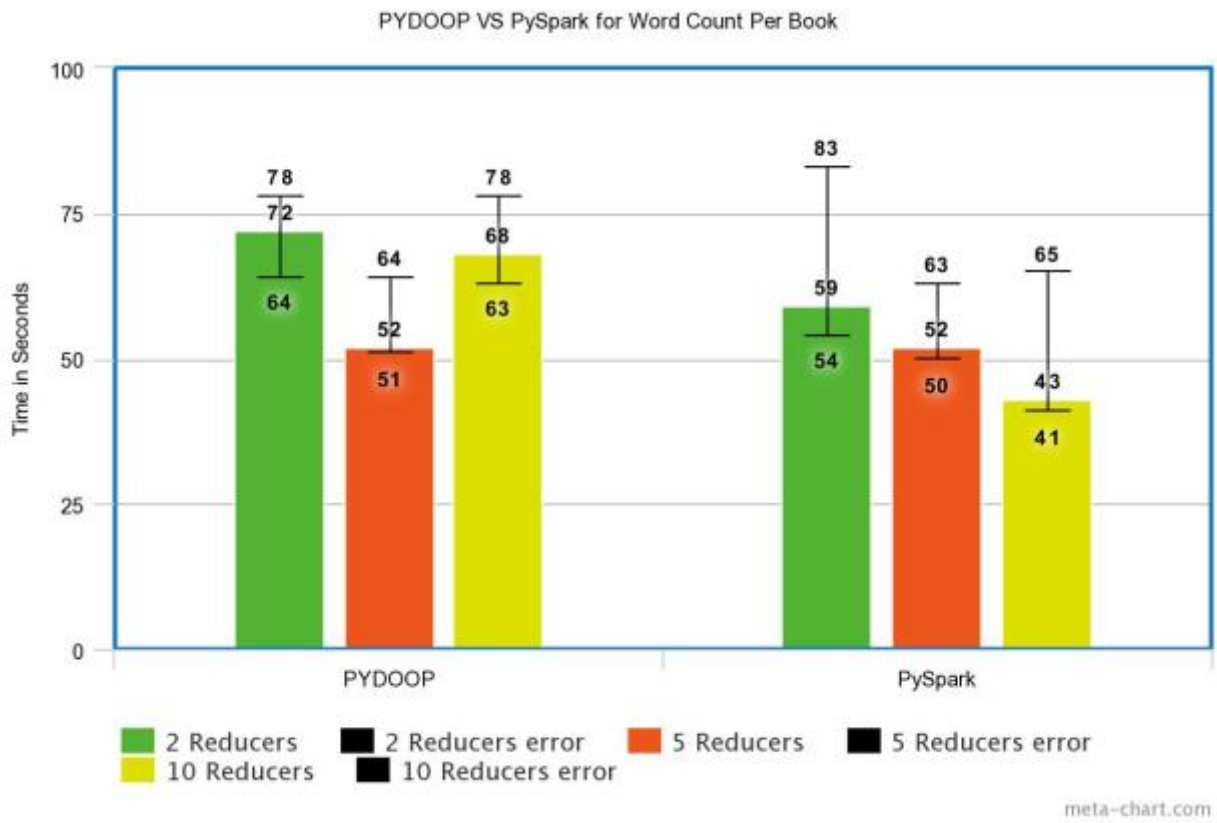

Fig 1. Time Taken by Part – 1

Part-2

No. of Executors	5	10	15
Avg Time Taken (5 readings) (in seconds)	102	98	99
Min Time Taken	96	91	94
Max Time Taken	114	118	112

Time Taken by Part -2

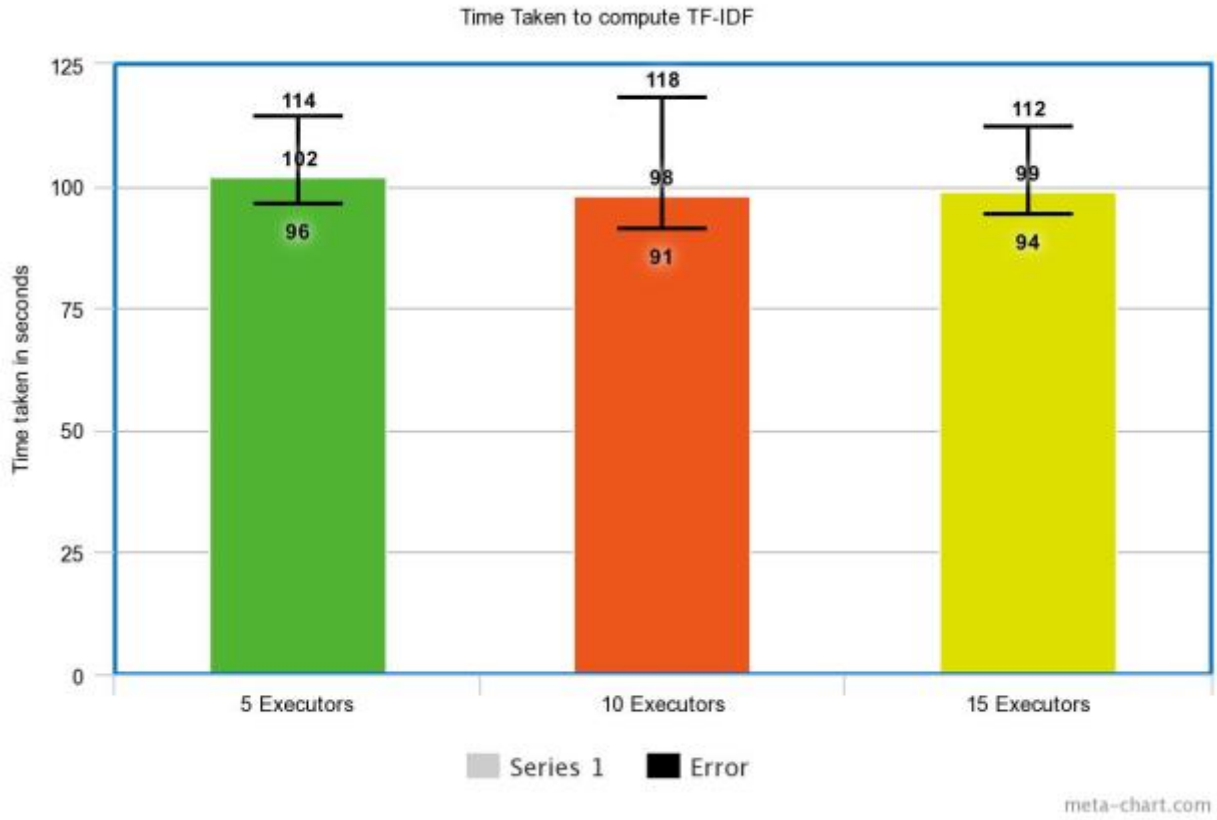

Fig 2. Time taken by Part – 2

Part-3

No. of Executors	5	10	15
Avg Time Taken (5 readings) (in seconds)	263	248	272
Min Time Taken	241	241	289
Max Time Taken	311	312	341

Time Taken by Part 3 for Only Word2Vec model
without finding synonyms for 10 words

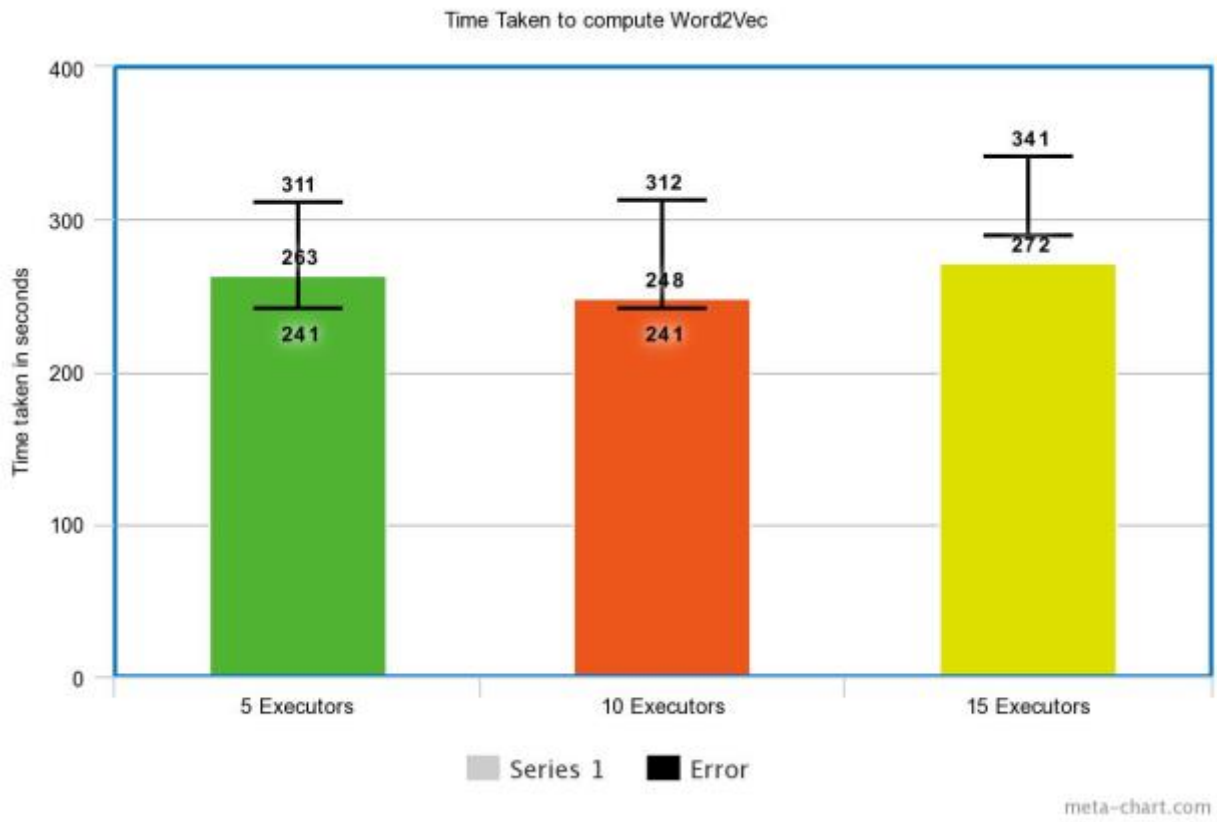

Fig 3. Time Taken by Part -3

No. of Executors	5	10	15
Avg Time Taken (5 readings) (in seconds)	449	454	431
Min Time Taken	421	411	422
Max Time Taken	463	456	438

Time Taken by Part 3 with Word2Vec model
along with finding synonyms for 10 words
with highest tf-idf and synonyms

Findings:

If we observe from above tables and bar plots we can notice that initially when the number of executors are increased we can see the time taken by the programs are reduced, but just only by varying the executors we can't see much difference, to see the difference in performance we have to vary different parameters like executor cores executor memory and by computing average running times the configuration with higher performance can be chosen for the spark job.

a. The performance of spark is not directly proportional to the no of executors it's dependent on other configurations as well
b. The performance of Spark Is better than what we obtained in map-reduce framework using pydoop.

The ideal no of executors, executor cores and executor memory can be obtained by performing multiple runs of the code and computing the average time, whichever configuration provides better time, it can be chosen as ideal configuration.

Why we need to compute ideal configuration for any spark job?

Because this would prevent the starvation of other execution processes and helps in effective utilization of available resources.

By caching the RDD's and DataFrames effectively we can improve the performance of the Spark Jobs. We can also persist the RDD's and DataFrames onto disk also.

Conclusion

Learnings from above assignment:

- The effective usage of Spark Cluster Computing model by leveraging on top of Hadoop distributed framework.
- An insight into how text processing problems containing large data sets can be solved used big data analytics.
- The significance of TF-IDF model in determining the importance of a word.
- The usage of word2vec model to determine co-existing words and synonyms.

Future Work:

- The above assignment can be extended to compute a predictive model such as Naïve Bayes classifier using the generated TF-IDF model.
- The TF-IDF model can be further extended to generate the text suggestions for a document search engine

References:

1. http://pstl.cs.uh.edu/resources/whale
2. https://en.wikipedia.org/wiki/Tf%E2%80%93idf