Report Homework 1 Parallel Processing and Data Analytics EECE 5645

002199302 Akanksha Agnihotri

Question 0 :

Answer: After running the command it gives us the output of the python program TextAnalyser.py , along it gives us the details of the various command that we can use to help us get the necessary output of the command . It gives us the input mode which is a positional argument and list of operational arguments .

```
(/work/courses/EECE5645/conda/EECE5645) [agnihotri.ak@login-00 Files]$ python TextAnalyzer.py --help
usage: TextAnalyzer.py [-h] [--master MASTER] [--N N]
                       [--simple_words SIMPLE_WORDS]
                       {SEN,WRD,UNQ,TOP20,DFF,DCF} input

Text Analysis via the Dale Chall Formula

positional arguments:
  {SEN,WRD,UNQ,TOP20,DFF,DCF}
                        Mode of operation
  input                 Text file to be processed. This file contains text
                        over several lines, with each line corresponding to a
                        different sentence.

optional arguments:
  -h, --help            show this help message and exit
  --master MASTER       Spark Master (default: local[20])
  --N N                 Number of partitions to be used in RDDs containing
                        word counts. (default: 20)
  --simple_words SIMPLE_WORDS
                        File containing Dale Chall simple word list. Each word
                        appears in one line. (default:
                        DaleChallEasyWordList.txt)
```

The portion of the code due to which it got printed is:

```
parser = argparse.ArgumentParser(description = 'Text Analysis via the Dale Chall Formula',formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('mode', help='Mode of operation',choices=['SEN','WRD','UNQ','TOP20','DFF','DCF'])
parser.add_argument('input', help='Text file to be processed. This file contains text over several lines, with each line corresponding to a different sentence.')
parser.add_argument('--master',default="local[20]",help="Spark Master")
parser.add_argument('--N',type=int,default=20,help="Number of partitions to be used in RDDs containing word counts.")
parser.add_argument('--simple_words',default="DaleChallEasyWordList.txt",help="File containing Dale Chall simple word list. Each word appears in one line.")
args = parser.parse_args()

sc = SparkContext(args.master, 'Text Analysis')
sc.setLogLevel('warn')
```

It displays information about the command line options and arguments that the script supports.

The information displayed may include:

A description of the purpose of the script, A list of required and optional arguments

A description of each argument, including its purpose and the expected format of the input

Examples of how to use the script. This information can be generated manually from script, or automatically by a library such as argparse.

It describes the format on which the input needs to be given in terms of choosing an option for the N partition , input files , options to choose , simple words . Hence it describes the use cases of the positional arguments.

Question 1:

Answer:

Part A

(i) Function definitions :

```
def strip_non_alpha(s):
    """ Remove non-alphabetic characters from the beginning and end of a string.

    E.g. ',1what?!"' should become 'what'. Non-alphabetic characters in the middle
    of the string should not be removed. E.g. "haven't" should remain unaltered."""

    while len(s)>0 and not s[-1].isalpha():
        s=s[:-1]
    while len(s)>0 and not s[0].isalpha():
        s=s[1:]
    return s
```

(ii) Test cases and Outputs :

Test cases :

```
def test_strip_non_alpha():
    assert strip_non_alpha("?hello world!") == "hello world"
    assert strip_non_alpha("HELLO ' World3") == "HELLO ' World"
    assert strip_non_alpha("!!@#$%^&*()") == ""
    assert strip_non_alpha("")==""
```

Outputs:

When you call the test_strip_non_alpha in the main function it test the above cases and tells if all test cases were passed or not

```
(/work/courses/EECE5645/conda/EECE5645) [agnihotri.ak@login-01 EECE5645]$ sr5645
(/work/courses/EECE5645/conda/EECE5645) [agnihotri.ak@c0508 EECE5645]$ python helpers.py
All cases of test_strip_non_alpha passed!
All cases of test_is_inflection_of is  passed!
All cases of test_same is passed!
All cases of test_find_match passed!
```

Part B :

i) Function definition:

```
def same(s1,s2):
    "Return True if one of the input strings is the inflection of the other."
    return(is_inflection_of(s1,s2) or is_inflection_of(s2,s1))
```

```
def is_inflection_of(s1,s2):
    """ Tests if s1 is a common inflection of s2.

    The function first (a) converts both strings to lowercase and (b) strips
    non-alphabetic characters from the beginning and end of each string.
    Then, it returns True if the two resulting two strings are equal, or
    the first string can be produced from the second by adding the following
    endings:
    (a) 's
    (b) s
    (c) es
    (d) ing
    (e) ed
    (f) d
    """

    s1=s1.lower()
    s2=s2.lower()
    s1=strip_non_alpha(s1)
    s2=strip_non_alpha(s2)

    if s1==s2 :
        return True
    else:
        list=["'s","s","es","ing","ed","d"]
        for end in list:
            if s2+end == s1:
                return True
        return False
```

```
def find_match(word,word_list):
    """Given a word, find a string in a list that is "the same" as this word.

    Input:
    - word: a string
    - word_list: a list of stings

    Return value:
    - A string in word_list that is "the same" as word, None otherwise.

    The string word is 'the same' as some string x in word_list, if word is the inflection of x,
    ignoring cases and leading or trailing non-alphabetic characters.
    """

    if len(word) == 0 or len(word_list) == 0:
        return None
    for w in word_list:
        if same(word, w):
            return w
    return None
```

ii) Test Cases and Outputs:

```
def test_is_inflection_of():
    assert is_inflection_of("hello", "hello") == True
    assert is_inflection_of("hello", "hell") == False
    assert is_inflection_of("hellos", "hello") == True
    assert is_inflection_of("helloes", "hello") == True
    assert is_inflection_of("helloing", "hello") == True
    assert is_inflection_of("helloed", "hello") == True
    assert is_inflection_of("hellod", "hello") == True
    assert is_inflection_of("hello's", "hello") == True
    assert is_inflection_of("","") == True

def test_same():
    assert same("hello", "hello") == True
    assert same("hellod", "hello") == True
    assert same("hello", "helloing") == True
    assert same("", "") == True
    assert same("", "a") == False

def test_find_match():
    assert find_match("hello", ["world", "hello", "HELLO"]) == "hello"
    assert find_match("Hello", ["world", "hello", "HELLO"]) == "hello"
    assert find_match("hello", []) == None
    assert find_match("", []) == None
```

Outputs : When you call the test_is_inflection_of , test_same, test_find_match in the main function it test the above cases and tells if all test cases were passed or not

```
(/work/courses/EECE5645/conda/EECE5645) [agnihotri.ak@login-01 EECE5645]$ sr5645
(/work/courses/EECE5645/conda/EECE5645) [agnihotri.ak@c0508 EECE5645]$ python helpers.py
All cases of test_strip_non_alpha passed!
All cases of test_is_inflection_of is  passed!
All cases of test_same is passed!
All cases of test_find_match passed!
```

Question 2:

Answer:

Part A)

i) Function definition of count_sentences and count_words

```
def count_sentences(rdd):
    """ Count the sentences in a file.

    Input:
    - rdd: an RDD containing the contents of a file, with one sentence in each element.


    Return value: The total number of sentences in the file.
    """

    return rdd.count()
```

```
def count_words(rdd):
    """ Count the number of words in a file.

    Input:
    - rdd: an RDD containing the contents of a file, with one sentence in each element.


    Return value: The total number of words in the file.
    """

    return len(rdd.split())
```

Part B)

Defining the rdd

```
sc = SparkContext(args.master, 'Text Analysis')
rdd = sc.textFile(args.input).repartition(args.N)
```

i) Main Definition:

```
if args.mode == 'SEN':
    num_sentences = count_sentences(rdd)
    print("Number of sentences:", num_sentences)
```

ii) By using the MobyDick.txt file the output is as follows :
Number of sentences: 10304

Part C)

i) Main Definition:

```
elif args.mode == 'WRD':
    num_words = rdd.flatMap(lambda line: line.split()).count()
    print("Number of words:", num_words)
```

ii) By using MobyDick.txt file the output is as follows:
Number of words: 215502

Part D)

Answer : The variable loaded is defined by sc in the program . Hence this loads a textfile MobyDick.txt within the rdd . Hence this the data/variable stored on the driver is the textfile contents that are pickable and stored within an rdd .

```
sc = SparkContext(args.master, 'Text Analysis')
rdd = sc.textFile(args.input).repartition(args.N)
```

The above two line tells us what is stored within the drivers .

The function such as the split() and lambda are sent to the workers so that the transformations performed by flatmap over there is stored in form an rdd as the variables within the workers

Also another rdd is formed within workers when we call action function .

Hence ,the communication between the driver and the worker is that it sends back the results that gets performed after applying the count function . So after performing the action by .count() it sends back result to the driver program . Hence the driver stores this value as an integer and then we can print this output

Question 3:

Part A) Function definitions :

```
def compute_counts(rdd,numPartitions=10):
    """ Produce an rdd that contains the number of occurences of each word in a file.

    Each word in the file is converted to lowercase and then stripped of leading and trailing non-alphabetic
    characters before its occurences are counted.

    Input:
    - rdd: an RDD containing the contents of a file, with one sentence in each element.


    Return value: an RDD containing pairs of the form (word,count), where word is is a lowercase string,
    without leading or trailing non-alphabetic characters, and count is the number of times it appears
    in the file. The returned RDD should have a number of partitions given by numPartitions.

    """
    # Reference taken from lecture 2,3,4,ppt taught by proffessor
    rdd2 = rdd.flatMap(lambda rdd: rdd.split())\
              .map(lambda rdd: to_lower_case(rdd))\
              .map(lambda rdd :strip_non_alpha(rdd))
    newrdd=rdd2.map(lambda word: (word, 1))\
             .reduceByKey(lambda x, y: x + y,numPartitions)
    return newrdd
```

Part B)

```
elif args.mode == 'UNQ':
    no_of_uniquewords=compute_counts(rdd,args.N)
    print("Number of unique words : ",no_of_uniquewords.count())
```

i)

ii) Applying this on the MobyDick.txt file

Answer: Number of unique words : 18622

Part C)

i)

```
elif args.mode =='TOP20':
    #Reference from proffesor's lecture 3 and 4 ppt .
    top20 = compute_counts(rdd,args.N).sortBy(lambda pair:pair[1],ascending = False).take(20)
    print("The top 20 records :",top20)
```

ii) Applying this over the MobyDick.txt file

Answer: top 20 :

The top 20 records : [('the', 14429), ('of', 6586), ('and', 6416), ('a', 4700), ('to', 4597), ('in', 4165), ('that', 2990), ('his', 2530), ('it', 2419),('i', 1989), ('but', 1818), ('he', 1777), ('as', 1741), ('is', 1722), ('with', 1722), ('was', 1644), ('for', 1617), ('all', 1509), ('this', 1394),('at',1318)]

Part D) Answers:

The driver consist of each sentences in each element when rdd reads a textfile MobyDick.txt considering the partition N which is given in the input

The workers consist of the values after implementing the flatmap and map functions as it transforms the data by converting them into lowercase and further implementing the strip_non_alpha functions and word over sentences that were sent from the driver . The functions such as to_lower_case and strip_non_alpha and split were sent to the rdd in the workers so that they can use transform like flatmap and map to get the values .

The new rrd2 also transforms its using map and inner function within it and store in workers.

The data that is transferred between from the workers to the drivers is the result obtained after implementing the action like reduceByKey after transforming under rdd2 that returns the unique word and their count along with the number of partitions . The workers sends this newrdd formed to the driver where in driver we perform count action to calculate the unique words .

Question 4 :

Part A) Function definition

```
def count_difficult_words(counts,easy_list):
    """ Count the number of difficult words in a file.

    Input:
    - counts: an RDD containing pairs of the form (word,count), where word is a lowercase string,
    without leading or trailing non-alphabetic characters, and count is the number of times this word appears
    in the file.
    - easy_list: a list of words deemed 'easy'.
    Return value: the total number of 'difficult' words in the file represented by RDD counts.

    A word should be considered difficult if is not the 'same' as a word in easy_list. Two words are the same
    if one is the inflection of the other, when ignoring cases and leading/trailing non-alphabetic characters.
    """
    #print(type(easy_list))
    return counts.filter(lambda a: find_match(a[0], easy_list) is None)\
                 .map(lambda a: a[1]).sum()
```

Part B)

```
elif args.mode =='DFF':

    counts=compute_counts(rdd,args.N)
    simple_words = create_list_from_file(args.simple_words)
    no_of_difficultwords = count_difficult_words(counts,simple_words)
    print("The number of difficult words are :",no_of_difficultwords)
```

I)

II) When we apply it over MobyDick.txt file we get
Answer: The number of difficult words are : 44366
(Refer the Addendum attached in report to check correctness of the code)

Part C)

Answer : The driver consist of the (words,counts) that we get after computing the compute_counts functions for the overall textfile . It also includes the simple_word list file that needs to compare and checked in order to get the difficult words

The worker consist of results that are obtained after implementation of the Filter , find_map function and map commands . The find map function maps key value pairs. This result is the stored in a new rdd within worker since we are calling the action function sum (). All the transform functions such as find_map , filter and map commands are sent to workers from the drivers .

The data that is transferred from the workers to the drivers include the results that are obtained after implementation of sum and counts . It gives the number of difficult words . Hence this is shared to the driver .

Part D)

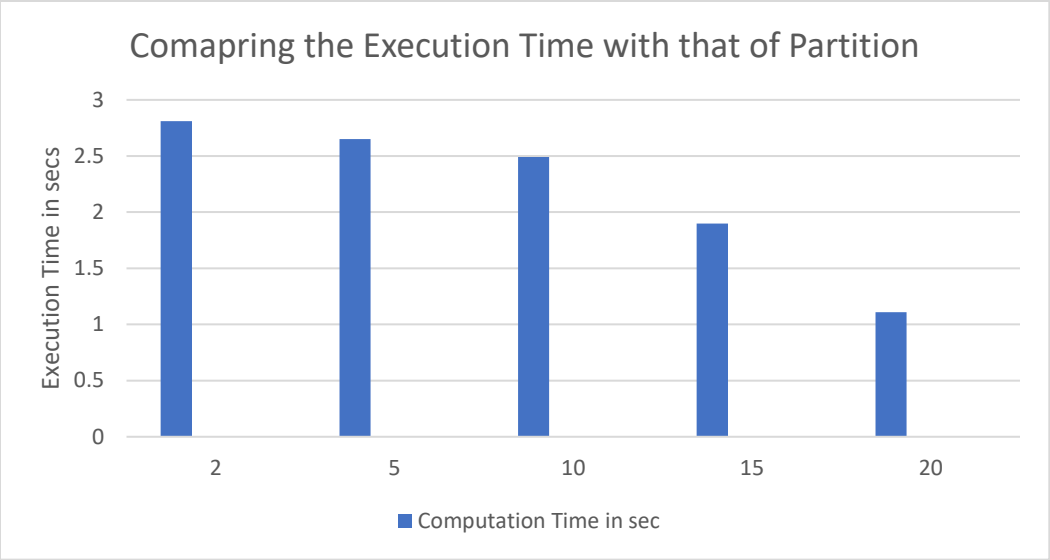
```
elif args.mode=='DCF':
    num_sentences = count_sentences(rdd)
    num_words = rdd.flatMap(lambda line: line.split()).count()
    counts=compute_counts(rdd,args.N)
    simple_words = create_list_from_file(args.simple_words)
    no_of_difficultwords = count_difficult_words(counts,simple_words)
    a= no_of_difficultwords/num_words
    b=num_words/num_sentences
    c = float((0.1579*(a*100))+(0.0496*(b)))
    print(" The Dale-Chall Formula is : ",c)
```

i)

ii) When we apply it over the MobyDick.txt file we get:
Answer:: The Dale-Chall Formula is : 4.28808584915786
(Refer the Addendum attached in report to check correctness of the code)

Question 5:

Answer: Plot for the N partitions [2,5,20,25,20] vs the execution time for running Dall Chall Formula



As seen from the graph the total time decreases as we see perform more partitions or increase the size of N. Since more number of partition means more workers are there to perform the work, hence it distributes the time faster and we are able to see the decrease in time .

Question 6:

Folder name	Name of the files	Dall Chall Formula	Size of each File	Average of Dall Chall Formula
Books	Frankenstein.txt	4.300616635066379	409 KB	4.22
	JaneEyre.txt	3.727418443349661	1000KB	
	MobyDick.txt	4.441295053712807	1.16 MB	
	PrideAndPrejudice.txt	3.7116405761276354	723 KB	
	TheOdyssey.txt	5.596993277518495	717 KB	
	TheStoryOfTheStone.txt	4.408975956020532	993 KB	
	WarAndPeace.txt	3.666112531103076	3.05 MB	
Movies	PiratesOfTheCarbibbean.txt	3.142496747023799	82.7 KB	3.14
News	NYT1.txt	6.710372450561032	8.79 KB	7
	NYT2.txt	7.691630871212122	1.24 KB	
	NYT3.txt	5.356221532091098	5.94 KB	

	WSJ1.txt	7.936376313594663	1.36 KB	
	WSJ2.txt	7.330955555555557	897 Bytes	

Answer: Table for every files in the data and their corresponding Dall Chall formula along with the size of each file.

The observation of the above file can be explained as below.

Its observed that News have the largest Dall chall formula as average is 7 and the size of the overall folder is also less means no of word and sentences were also less that is why its more

Movies has lowest value dall chall formula which is 3.14 and has only 1 big file to evaluate.

Books have the average of 4.22 and the overall size is more as compared to other folders hence more number of words and sentences are expected that is why is its more than the News folder .

It is observed that WSJ1.txt has the highest Dall Chall formula value and PiratesOfTheCaribibbean.txt has the lowest Dall Chall Formula value .

Correctness:

In order to see the correctness of the code please find the below output of the Addendum implemented on the fireandice.txt file for the code.

```
(/work/courses/EECE5645/conda/EECE5645) [agnihotri.ak@c0511 EECE5645]$ python TextAnalyzer.py SEN fireandice.txt
23/02/09 21:09:16 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-jav
plicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Number of sentences: 9
Total execution time: 1.1920928955078125e-06sec

(/work/courses/EECE5645/conda/EECE5645) [agnihotri.ak@c0511 EECE5645]$ python TextAnalyzer.py WRD fireandice.txt
23/02/09 21:09:35 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java cl
plicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Number of words: 51
Total execution time: 1.6689300537109375e-06sec

(/work/courses/EECE5645/conda/EECE5645) [agnihotri.ak@c0511 EECE5645]$ python TextAnalyzer.py UNQ fireandice.txt --N 20
23/02/09 21:10:20 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java cla
plicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Number of unique words :  41
Total execution time: 1.1920928955078125e-06sec
```

```
(/work/courses/EECE5645/conda/EECE5645) [agnihotri.ak@c0511 EECE5645]$ python TextAnalyzer.py TOP20 fireandice.txt --N 20
23/02/09 21:10:49 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where ap
plicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
The top 20 records : [('i', 3), ('say', 3), ('of', 2), ('to', 2), ('some', 2), ('fire', 2), ('in', 2), ('ice', 2), ('think', 1), ('wou
ld', 1), ('and', 1), ('it', 1), ('twice', 1), ('also', 1), ('desire', 1), ('hold', 1), ('will', 1), ("i've", 1), ('had', 1), ('perish'
, 1)]
```

```
(/work/courses/EECE5645/conda/EECE5645) [agnihotri.ak@c0511 EECE5645]$ python TextAnalyzer.py DFF fireandice.txt --N 20 --simple_words
 DaleChallEasyWordList.txt
23/02/09 21:12:01 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where ap
plicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
The number of difficult words are : 3


The words are : (perish,destruction,suffice)
```

```
(/work/courses/EECE5645/conda/EECE5645) [agnihotri.ak@c0511 EECE5645]$ python TextAnalyzer.py DCF fireandice.txt
23/02/09 21:13:12 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where a
plicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
 The Dale-Chall Formula is :  1.2098901960784314
Total execution time: 1.1920928955078125e-06sec
```