RICE UNIVERSITY

COMP543 Lab 5: Introduction to Spark

COMP543 Instructors

Last edited: September 25, 2018

The purpose of this lab is to get some experience with Spark. You might use the lecture Notes (on Piazza) as a reference. Here are the two subtasks.

1 Getting Setup

To complete this lab, you'll need to have access to either a spark cluster or a local instance of spark. A docker image is provided for you at gvacaliuc/spark. If you are not using docker, you can try installing and downloading Spark here: http://spark.apache.org/downloads.html. Eventually, we'll have a lab to get you setup on Amazon Web Services so that you'll be able to run and connect to a spark cluster in the cloud. If you don't have docker setup and you're having trouble getting Spark installed, drop a note to Gabe (gv8@rice.edu).

The following sections will assume that you have the ability to run code with a spark context available at `sc`.

Please note that if you're running the code via Docker you **will not** be able to use the `s3` links to our data files. Rather, please use the local text files provided with the lab.

2 Running WordCount on Spark

The first task we'll perform is counting word frequencies with Spark. This is usually the first task when performing Natural Language Processing, and a common "Hello, World" task for Hadoop and Spark.

If you're not working off of our provided notebook, copy the following function definition into your python terminal or notebook:

```
def countWords (fileName):
  textfile = sc.textFile(fileName)
  lines = textfile.flatMap(lambda line: line.split(" "))
  counts = lines.map (lambda word: (word, 1))
  aggregatedCounts = counts.reduceByKey (lambda a, b: a + b)
  return aggregatedCounts.top (200, key=lambda p : p[1])
```

This method counts the occurrences of the most common 200 words in our textfile. First, we load a text file into spark line by line with `sc.textFile`. Then, we perform a flat map (in which each lambda produces an iterable which is eventually flattened at the end) to get an RDD of words. Then, we map each `word` to a tuple of `(word, 1)`, and reduce (aggregate) by key to get overall word counts.

3 Completing a Spark Program

In this task, we're giving you an almost-complete code that analyzes a corpus of real text documents. Your task us to complete my code.

You can start by taking a look at the data set. Check out the file provided with this lab (`20-news-same-line-small.txt`).

There are 19997 lines in this file, each corresponding to a different text document. The goal here is to build, as an RDD, a dictionary. The dictionary will have as its key a number from 0 to 19,999 (this is a rank) and the value is the word corresponding to that rank. The words will be ranked according to their frequency in the corpus, with 0 being the most frequent, 19,999 being the 20-thousandth-most frequent.

Start by reading the code (included in the notebook and provided below). Note that no computation is performed until you enter the line that tries to collect the results (using the call to top), at which point Spark goes off, performs all of the computations, and tries collect the results into topWords. At that point, `topWords` is a local Python variable (as opposed to a reference to an RDD). In the Python shell, you can manipulate it, change it, and print it, just as you would any other local variable.

```
import re
import numpy as np

# load up all of the 19997 documents in the corpus
corpus = sc.textFile ("s3://risamyersbucket/comp543_Lab5/20_news_same_line.txt")

# each entry in validLines will be a line from the text file
validLines = corpus.filter(lambda x : 'id' in x)

# now we transform it into a bunch of (docID, text) pairs
keyAndText = validLines.map(lambda x : (x[x.index('id="') + 4 : x.index('" url=')],
↪  x[x.index('">') + 2:]))

# now we split the text in each (docID, text) pair into a list of words
# after this, we have a data set with (docID, ["word1", "word2", "word3", ...])
# we have a bit of fancy regular expression stuff here to make sure that we do not
# die on some of the documents
regex = re.compile('[^a-zA-Z]')
```

```
keyAndListOfWords = keyAndText.map(lambda x : (str(x[0]), regex.sub(' ',
↪  x[1]).lower().split()))

# now get the top 20,000 words... first change (docID, ["word1", "word2", "word3",
↪  ...])
# to ("word1", 1) ("word2", 1)...
allWords = keyAndListOfWords.flatMap(lambda x: ((j, 1) for j in x[1]))

# now, count all of the words, giving us ("word1", 1433), ("word2", 3423423), etc.
allCounts = allWords.reduceByKey (lambda a, b: a + b)

# and get the top 20,000 words in a local array
# each entry is a ("word1", count) pair
topWords = allCounts.top (20000, lambda x : x[1])

# and we'll create a RDD that has a bunch of (word, dictNum) pairs
# start by creating an RDD that has the number 0 thru 20000
# 20000 is the number of words that will be in our dictionary
twentyK = sc.parallelize(range(20000))

# now, we transform (0), (1), (2), ... to ("mostcommonword", 0) ("nextmostcommon",
↪  1), ...
# the number will be the spot in the dictionary used to tell us where the word is
↪  located
# HINT: make use of topWords in the lambda that you supply
# YOUR CODE HERE
dictionary = twentyK.map (??????????????????????)

# finally, print out some of the dictionary, just for debugging
dictionary.top (10)
```

Once you've made it to the question marks, write an appropriate lambda that is going to attach the correct word to each number, putting the results into an RDD. Replace the question marks with your lambda. The key of the tuple you create and put into the RDD via the lambda should be the word; the value is the index (the number from 0 to 19,999). Your lambda will refer to the local variable topWords.

After you run `dictionary.top(10)` you can get checked off! Note that this will return the top 10 words by name (the key), and not the index (the value).