95-869: Big Data and Large-Scale Computing Homework 1

Introduction to PySpark and RDDs

The volume of unstructured text in existence is growing dramatically, and Spark is an excellent tool for analyzing this type of data. In this homework, we will write code that calculates the most common words in the Complete Works of William Shakespeare (http://www.gutenberg.org/ebooks/100) retrieved from Project Gutenberg (http://www.gutenberg.org/wiki/Main_Page).

This could also be scaled to find the most common words in Wikipedia.

In this homework, we will cover:

- *Part 1 (25 Points):* Creating a base RDD and pair RDDs
- *Part 2 (25 Points):* Counting with pair RDDs
- *Part 3 (25 Points):* Finding unique words and a mean value
- *Part 4 (25 Points):* Apply word count to a file

Submission Instructions:

You will submit both a zipped file on Canvas and a hard copy at the beginning of the class by the deadline date.

Rename the notebook from "hw1_pyspark_rdd_student.ipynb" to "andrewid_hw1_pyspark_rdd_student.ipynb" where "andrewid" is your actual Andrew ID. Complete the assignment, execute all cells in the completed notebook, and make sure all results show up. Export the contents of the notebook by choosing "File > Download as > HTML" and saving the resulting file as "andrewid_hw1_pyspark_rdd_student.html" Place the two files "andrewid_hw1_pyspark_rdd_student.ipynb" and "andrewid_hw1_pyspark_rdd_student.html" in a folder, zip the folder to a zipped file named "andrewid_hw1.zip" and submit it to Canvas by the deadline. In addition, print the HTML file and submit the hard copy at the beginning of the class on the date of the submission.

Note that, for reference, you can look up the details of the relevant Spark methods in Spark's Python API (https://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD) and the relevant NumPy methods in the NumPy Reference (http://docs.scipy.org/doc/numpy/reference/index.html)

Part 1: Creating a base RDD and pair RDDs

In this part, we will explore creating a base RDD with `parallelize` and using pair RDDs to count words.

Let's first start by creating the SparkContext

In [ ]:
```
import sys
sys.path.append("/opt/packages/spark/latest/python/lib/py4j-0.10.4-src.zip")
sys.path.append("/opt/packages/spark/latest/python/")
sys.path.append("/opt/packages/spark/latest/python/pyspark")
from pyspark import SparkConf, SparkContext
sc = SparkContext()
sc
```

(1a) Create a base RDD

We'll start by generating a base RDD by using a Python list and the `sc.parallelize` method. Then we'll print out the type of the base RDD.

In [ ]:
```
wordsList = ['cat', 'elephant', 'rat', 'rat', 'cat']
wordsRDD = sc.parallelize(wordsList, 4)
# Print out the type of wordsRDD
print (type(wordsRDD))
```

(1b) Pluralize and test

Let's use a `map()` transformation to add the letter 's' to each string in the base RDD we just created. We'll define a Python function that returns the word with an 's' at the end of the word. Please replace `<FILL IN>` with your solution. After you have defined `makePlural` you can run the third cell which contains a test. If you implementation is correct it will print `1 test passed`.

This is the general form that exercises will take, except that no example solution will be provided. Exercises will include an explanation of what is expected, followed by code cells where one cell will have one or more `<FILL IN>` sections. The cell that needs to be modified will have `# TODO: Replace <FILL IN> with appropriate code` on its first line. Once the `<FILL IN>` sections are updated and the code is run, the test cell can then be run to verify the correctness of your solution. The last code cell before the next markdown section will contain the tests.

In [ ]:
```
# TODO: Replace <FILL IN> with appropriate code
def makePlural(word):
    """Adds an 's' to `word`.

    Note:
        This is a simple function that only adds an 's'.  No attempt is
 made to follow proper
        pluralization rules.

    Args:
        word (str): A string.

    Returns:
        str: A string with 's' added to it.
    """
    return <FILL IN>

print (makePlural('cat'))
```

(1c) Apply `makePlural` to the base RDD

Now pass each item in the base RDD into a map() (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.map) transformation that applies the `makePlural()` function to each element. And then call the collect() (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.collect) action to see the transformed RDD.

In [ ]:
```
# TODO: Replace <FILL IN> with appropriate code
pluralRDD = wordsRDD.map(<FILL IN>)
print (pluralRDD.collect())
```

(1d) Pass a `lambda` function to `map`

`lambda` functions in Python are anonymous functions that are created for one-time use. Let's create the same RDD using a `lambda` function.

In [ ]:
```
# TODO: Replace <FILL IN> with appropriate code
pluralLambdaRDD = wordsRDD.map(lambda <FILL IN>)
print (pluralLambdaRDD.collect())
```

(1e) Length of each word

Now use `map()` and a `lambda` function to return the number of characters in each word. We'll `collect` this result directly into a variable.

In [ ]:
```
# TODO: Replace <FILL IN> with appropriate code
pluralLengths = (pluralRDD
                 <FILL IN>
                 .collect())
print (pluralLengths)
```

(1f) Pair RDDs

The next step in writing our word counting program is to create a new type of RDD, called a pair RDD. A pair RDD is an RDD where each element is a pair tuple `(k, v)` where `k` is the key and `v` is the value. In this example, we will create a pair consisting of `('<word>', 1)` for each word element in the RDD. We can create the pair RDD using the `map()` transformation with a `lambda()` function to create a new RDD.

In [ ]:
```
# TODO: Replace <FILL IN> with appropriate code
wordPairs = wordsRDD.<FILL IN>
print (wordPairs.collect())
```

Part 2: Counting with pair RDDs

Now, let's count the number of times a particular word appears in the RDD. There are multiple ways to perform the counting, but some are much less efficient than others.

A naive approach would be to `collect()` all of the elements and count them in the driver program. While this approach could work for small datasets, we want an approach that will work for any size dataset including terabyte- or petabyte-sized datasets. In addition, performing all of the work in the driver program is slower than performing it in parallel in the workers. For these reasons, we will use data parallel operations.

(2a) `groupByKey()` approach

An approach you might first consider (we'll see shortly that there are better ways) is based on using the groupByKey() (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.groupByKey) transformation. As the name implies, the `groupByKey()` transformation groups all the elements of the RDD with the same key into a single list in one of the partitions.

There are two problems with using `groupByKey()`:

- The operation requires a lot of data movement to move all the values into the appropriate partitions.
- The lists can be very large. Consider a word count of English Wikipedia: the lists for common words (e.g., the, a, etc.) would be huge and could exhaust the available memory in a worker.

Use `groupByKey()` to generate a pair RDD of type `('word', iterator)`.

In [ ]:
```
# TODO: Replace <FILL IN> with appropriate code
# Note that groupByKey requires no parameters
wordsGrouped = wordPairs.<FILL IN>
for key, value in wordsGrouped.collect():
    print ('{0}: {1}'.format(key, list(value)))
```

(2b) Use `groupByKey()` to obtain the counts

Using the `groupByKey()` transformation creates an RDD containing 3 elements, each of which is a pair of a word and a Python iterator.

Now sum the iterator using a `map()` transformation. The result should be a pair RDD consisting of (word, count) pairs.

In [ ]:
```
# TODO: Replace <FILL IN> with appropriate code
wordCountsGrouped = wordsGrouped.<FILL IN>
print (wordCountsGrouped.collect())
```

(2c) Counting using `reduceByKey`

A better approach is to start from the pair RDD and then use the reduceByKey() (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.reduceByKey) transformation to create a new pair RDD. The `reduceByKey()` transformation gathers together pairs that have the same key and applies the function provided to two values at a time, iteratively reducing all of the values to a single value. `reduceByKey()` operates by applying the function first within each partition on a per-key basis and then across the partitions, allowing it to scale efficiently to large datasets.

In [ ]:
```
# TODO: Replace <FILL IN> with appropriate code
# Note that reduceByKey takes in a function that accepts two values and
 returns a single value
wordCountsGrouped = wordPairs.reduceByKey(<FILL IN>)
print (wordCountsGrouped.collect())
```

(2d) All together

The expert version of the code performs the `map()` to pair RDD, `reduceByKey()` transformation, and `collect` in one statement.

In [ ]:
```
# TODO: Replace <FILL IN> with appropriate code
wordCountsCollected = (wordsRDD
                       <FILL IN>
                       .collect())
print (wordCountsCollected)
```

Part 3: Finding unique words and a mean value

(3a) Unique words

Calculate the number of unique words in `wordsRDD`. You can use other RDDs that you have already created to make this easier.

In [ ]:
```
# TODO: Replace <FILL IN> with appropriate code
uniqueWords = (<FILL IN>)
print (uniqueWords)
print (len(uniqueWords))
```

(3b) Mean using `reduce`

Find the mean number of words per unique word in `wordCounts`.

Use a `reduce()` action and the imported `add` function to sum the counts in `wordCounts` and then divide by the number of unique words. First `map()` the pair RDD `wordCounts`, which consists of (key, value) pairs, to an RDD of values.

In [ ]:
```
# TODO: Replace <FILL IN> with appropriate code
from operator import add
totalCount = (wordCounts
              .map(<FILL IN>)
              .reduce(<FILL IN>))
average = totalCount / float(<FILL IN>)
print (totalCount)
print (round(average, 2))
```

Part 4: Apply word count to a file

In this section we will finish developing our word count application. We'll have to build the `wordCount` function, deal with real world problems like capitalization and punctuation, load in our data source, and compute the word count on the new data.

(4a) `wordCount` function

First, define a function for word counting. You should reuse the techniques that have been covered in earlier parts of this assignment. This function should take in an RDD that is a list of words like `wordsRDD` and return a pair RDD that has all of the words and their associated counts.

In [ ]:
```
# TODO: Replace <FILL IN> with appropriate code
def wordCount(wordListRDD):
    """Creates a pair RDD with word counts from an RDD of words.

    Args:
        wordListRDD (RDD of str): An RDD consisting of words.

    Returns:
        RDD of (str, int): An RDD consisting of (word, count) tuples.
    """
    <FILL IN>
print (wordCount(wordsRDD).collect())
```

(4b) Capitalization and punctuation

Real world files are more complicated than the data we have been using thus far. Some of the issues we have to address are:

- Words should be counted independent of their capitialization (e.g., Spark and spark should be counted as the same word).
- All punctuation should be removed.
- Any leading or trailing spaces on a line should be removed.

Define the function `removePunctuation` that converts all text to lower case, removes any punctuation, and removes leading and trailing spaces. Use the Python re (https://docs.python.org/2/library/re.html) module to remove any text that is not a letter, number, or space. Reading `help(re.sub)` might be useful. If you are unfamiliar with regular expressions, you may want to review this tutorial (https://developers.google.com/edu/python/regular-expressions) from Google. Also, this website (https://regex101.com/#python) is a great resource for debugging your regular expression.

In [ ]:
```
# TODO: Replace <FILL IN> with appropriate code
import re
def removePunctuation(text):
    """Removes punctuation, changes to lower case, and strips leading an
d trailing spaces.

    Note:
        Only spaces, letters, and numbers should be retained.  Other cha
racters should should be
        eliminated (e.g. it's becomes its).  Leading and trailing spaces
 should be removed after
        punctuation is removed.

    Args:
        text (str): A string.

    Returns:
        str: The cleaned up string.
    """
    <FILL IN>
print (removePunctuation('Hi, you!'))
print (removePunctuation(' No under_score!'))
print (removePunctuation(' *      Remove punctuation then spaces  * '))
```

(4c) Load a text file

For the next part of this homework, we will use the Complete Works of William Shakespeare (http://www.gutenberg.org/ebooks/100) from Project Gutenberg (http://www.gutenberg.org/wiki/Main_Page). To convert a text file into an RDD, we use the `SparkContext.textFile()` method. We also apply the recently defined `removePunctuation()` function using a `map()` transformation to strip out the punctuation and change all text to lower case. Since the file is large we use `take(15)`, so that we only print 15 lines.

In [ ]:
```
# Just run this code

fileName = 'file:///pylon5/ci5619p/ahmaurya/shakespeare.txt'

shakespeareRDD = sc.textFile(fileName, 8).map(removePunctuation)
sampleLines = ('\n'.join(shakespeareRDD
                .zipWithIndex()  # to (line, lineNum)
                .map(lambda (l, num): '{0}: {1}'.format(num, l))  # to
 'lineNum: line'
                .take(15)))
print (sampleLines)
```

(4d) Words from lines

Before we can use the `wordcount()` function, we have to address two issues with the format of the RDD:

- The first issue is that that we need to split each line by its spaces. **Performed in (4d).**
- The second issue is we need to filter out empty lines. **Performed in (4e).**

Apply a transformation that will split each element of the RDD by its spaces. For each element of the RDD, you should apply Python's string split() (https://docs.python.org/2/library/string.html#string.split) function. You might think that a `map()` transformation is the way to do this, but think about what the result of the `split()` function will be.

> Note:
>
> - Do not use the default implementation of `split()`, but pass in a separator value. For example, to split `line` by commas you would use `line.split(',')`.

In [ ]:
```
# TODO: Replace <FILL IN> with appropriate code
shakespeareWordsRDD = shakespeareRDD.<FILL_IN>
shakespeareWordCount = shakespeareWordsRDD.count()
print (shakespeareWordsRDD.top(5))
print (shakespeareWordCount)
```

(4e) Remove empty elements

The next step is to filter out the empty elements. Remove all entries where the word is `''`.

In [ ]:
```
# TODO: Replace <FILL IN> with appropriate code
shakeWordsRDD = shakespeareWordsRDD.<FILL_IN>
shakeWordCount = shakeWordsRDD.count()
print (shakeWordCount)
```

(4f) Count the words

We now have an RDD that is only words. Next, let's apply the `wordCount()` function to produce a list of word counts. We can view the top 15 words by using the `takeOrdered()` action; however, since the elements of the RDD are pairs, we need a custom sort function that sorts using the value part of the pair.

You'll notice that many of the words are common English words. These are called stopwords. Use the `wordCount()` function and `takeOrdered()` to obtain the fifteen most common words and their counts.

In [ ]:
```
# TODO: Replace <FILL IN> with appropriate code
top15WordsAndCounts = <FILL IN>
print ('\n'.join(map(lambda (w, c): '{0}: {1}'.format(w, c), top15WordsA
ndCounts)))
```

The End