Word Count Assignment: Building a word count application

The volume of unstructured text in existence is growing dramatically, and Spark is an excellent tool for analyzing this type of data. In this assignment, we will write code that calculates the most common words in the Complete Works of William Shakespeare retrieved from Project Gutenberg.

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

During this assignment we will cover:

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

Note that for reference, you can look up the details of the relevant methods in:

- Spark's Python API

Part 1: Creating a base RDD and pair RDDs

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

(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.

```
>
wordsList = ['cat', 'elephant', 'rat', 'rat', 'cat']
wordsRDD = sc.parallelize(wordsList, 4)

# Print out the type of wordsRDD
print type(wordsRDD)
<class 'pyspark.rdd.RDD'>
```

(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.

Exercises will include an explanation of what is expected, followed by codes where they will have one or more <FILL IN> sections. The code that needs to be modified will have # TODO: Replace <FILL IN> with appropriate code on its first line.

```
>
# 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')
cats
```

(1c) Apply `makePlural` to the base RDD
Now pass each item in the base RDD into a `map()` transformation that applies the `makePlural()` function to each element. And then call the `collect()` action to see the transformed RDD.

```
>
# TODO: Replace <FILL IN> with appropriate code
pluralRDD = wordsRDD.map(<FILL IN>)

print pluralRDD.collect()
['cats', 'elephants', 'rats', 'rats', 'cats']
```

(1d) Pass a `lambda` function to `map`
Let's create the same RDD using a `lambda` function.

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

print pluralLambdaRDD.collect()
['cats', 'elephants', 'rats', 'rats', 'cats']
```

(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.

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

print pluralLengths
[4, 9, 4, 4, 4]
```

(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.

```
>
# TODO: Replace <FILL IN> with appropriate code
wordPairs = wordsRDD.<FILL IN>

print wordPairs.collect()
[('cat', 1), ('elephant', 1), ('rat', 1), ('rat', 1), ('cat', 1)]
```

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()` 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).`

```
>
# 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))

rat: [1, 1]
elephant: [1]
cat: [1, 1]

sorted(wordsGrouped.mapValues(lambda x: list(x)).collect())
[('cat', [1, 1]), ('elephant', [1]), ('rat', [1, 1])]
```

(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.

```
>
# TODO: Replace <FILL IN> with appropriate code
wordCountsGrouped = wordsGrouped.<FILL IN>

print wordCountsGrouped.collect()
[('rat', 2), ('elephant', 1), ('cat', 2)]
```

(2c) Counting using `reduceByKey`
A better approach is to start from the pair RDD and then use the `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.

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

print wordCounts.collect()
[('rat', 2), ('elephant', 1), ('cat', 2)]
```

(2d) All together
The expert version of the code performs the `map()` to pair RDD, `reduceByKey()` transformation, and `collect` in one statement.

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

print wordCountsCollected
[('rat', 2), ('elephant', 1), ('cat', 2)]

sorted(wordCountsCollected)
[('cat', 2), ('elephant', 1), ('rat', 2)]
```

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.

```
>
# TODO: Replace <FILL IN> with appropriate code
uniqueWords = <FILL IN>
print uniqueWords
3
```

(3b) Mean using `reduce`
Find the mean number of words per unique word in `wordCounts`.

Use a `reduce()` action 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.

```
>
# 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
5

print round(average, 2)
1.67
```

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.

```
>
# 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()
[(‘rat', 2), ('elephant', 1), ('cat', 2)]
```

(4b) Capitalization and punctuation
Real world files are more complicated than the data we have been using here. Some of the issues we have to address are:
• Words should be counted independent of their capitalization (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 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 from Google. Also, this website is a great resource for debugging your regular expression.

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

    Note:
        Only spaces, letters, and numbers should be retained.  Other characters 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!')
hi you

print removePunctuation(' No under_score!')
no underscore

print removePunctuation(' *      Remove punctuation then spaces  * ‘)
remove punctuation then spaces
```

(4c) Load a text file
For the next part of this assignment, we will use the Complete Works of William Shakespeare from Project Gutenberg. 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. Download `shakespeare.txt` file and store it in your Spark directory. Since the file is large we use `take(15)`, so that we only print 15 lines.

```
>
# Just run this code
fileName = "shakespeare.txt"
```

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

(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() 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 implemenation of `split()`, but pass in a separator value. For example, to split `line` by commas you would use `line.split(',')`.

```
>
# TODO: Replace <FILL IN> with appropriate code
shakespeareWordsRDD = shakespeareRDD.<FILL_IN>
shakespeareWordCount = shakespeareWordsRDD.count()

print shakespeareWordsRDD.top(5)
[u'zwaggerd', u'zounds', u'zounds', u'zounds', u'zounds']

print shakespeareWordCount
946354
```

(4e) Remove empty elements
The next step is to filter out the empty elements. Remove all entries where the word is `' '`.

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

print shakeWordCount
901109
```

(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. In a later assignment, we will see how to eliminate them from the results. Use the `wordCount()` function and `takeOrdered()` to obtain the fifteen most common words and their counts.

```
>
# TODO: Replace <FILL IN> with appropriate code
top15WordsAndCounts = <FILL IN>

print '\n'.join(map(lambda (w, c): '{0}: {1}'.format(w, c), top15WordsAndCounts))
the: 27645
and: 26733
i: 20683
to: 19198
of: 18180
a: 14613
you: 13650
my: 12480
that: 11122
in: 10967
is: 9598
not: 8725
for: 8245
with: 7996
me: 7768
```