Word Count Lab: Building a word count application

This lab will build on the techniques covered in the Spark tutorial to develop a simple 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 lab, 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 on the Internet.

During this lab 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 (https://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD)

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.

In [1]:
```
wordsList = ['cat', 'elephant', 'rat', 'rat', 'cat']
wordsRDD = sc.parallelize(wordsList, 4)
# Print out the type of 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. If you have trouble, the next cell has the 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 [2]:
```
# 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
        pluralization rules.

    Args:
        word (str): A string.

    Returns:
        str: A string with 's' added to it.
    """
    return word + 's'
```

```
cats
```

In [3]:
```
# One way of completing the function
def makePlural(word):
    return word + 's'
```

```
cats
```

In [4]:
```
# Load in the testing code and check to see if your answer is correct
# If incorrect it will report back '1 test failed' for each failed test
# Make sure to rerun any cell you change before trying the test again
from test_helper import Test
# TEST Pluralize and test (1b)
```

```
1 test passed.
```

(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 [5]:
```
# TODO: Replace <FILL IN> with appropriate code
pluralRDD = wordsRDD.map(makePlural)
```

```
['cats', 'elephants', 'rats', 'rats', 'cats']
```

In [6]:
```
# TEST Apply makePlural to the base RDD(1c)
Test.assertEquals(pluralRDD.collect(), ['cats', 'elephants', 'rats', 'rats', 'ca
```

```
1 test passed.
```

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

Let's create the same RDD using a `lambda` function.

In [7]:
```
# TODO: Replace <FILL IN> with appropriate code
pluralLambdaRDD = wordsRDD.map(lambda word: word + 's')
```

```
['cats', 'elephants', 'rats', 'rats', 'cats']
```

In [8]:
```
# TEST Pass a lambda function to map (1d)
Test.assertEquals(pluralLambdaRDD.collect(), ['cats', 'elephants', 'rats', 'rats
```

```
1 test passed.
```

(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 [9]:
```
# TODO: Replace <FILL IN> with appropriate code
pluralLengths = (pluralRDD
                 .map(lambda word: len(word))
                 .collect())
```

```
[4, 9, 4, 4, 4]
```

In [10]:
```
# TEST Length of each word (1e)
Test.assertEquals(pluralLengths, [4, 9, 4, 4, 4],
```

```
1 test passed.
```

(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 [11]:
```
# TODO: Replace <FILL IN> with appropriate code
wordPairs = wordsRDD.map(lambda word: (word, 1))
```

```
[('cat', 1), ('elephant', 1), ('rat', 1), ('rat', 1), ('cat', 1)]
```

In [12]:
```
# TEST Pair RDDs (1f)
Test.assertEquals(wordPairs.collect(),
                  [('cat', 1), ('elephant', 1), ('rat', 1), ('rat', 1), ('cat',
```

```
1 test passed.
```

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 [13]:
```
# TODO: Replace <FILL IN> with appropriate code
# Note that groupByKey requires no parameters
wordsGrouped = wordPairs.groupByKey()
for key, value in wordsGrouped.collect():
```

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

In [14]:
```
# TEST groupByKey() approach (2a)
Test.assertEquals(sorted(wordsGrouped.mapValues(lambda x: list(x)).collect()),
                  [('cat', [1, 1]), ('elephant', [1]), ('rat', [1, 1])],
```

```
1 test passed.
```

(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 [15]:
```
# TODO: Replace <FILL IN> with appropriate code
wordCountsGrouped = wordsGrouped.map(lambda (k, v): (k, sum(v)))
```

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

In [16]:
```
# TEST Use groupByKey() to obtain the counts (2b)
Test.assertEquals(sorted(wordCountsGrouped.collect()),
                  [('cat', 2), ('elephant', 1), ('rat', 2)],
```

```
1 test passed.
```

(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 [22]:
```
# TODO: Replace <FILL IN> with appropriate code
# Note that reduceByKey takes in a function that accepts two values and returns
wordCounts = wordPairs.reduceByKey(lambda x, y: x + y)
print wordCounts.collect()
```

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

In [23]:
```
# TEST Counting using reduceByKey (2c)
Test.assertEquals(sorted(wordCounts.collect()), [('cat', 2), ('elephant', 1), ('
```

```
1 test passed.
```

(2d) All together

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

In [24]:
```
# TODO: Replace <FILL IN> with appropriate code
wordCountsCollected = (wordsRDD
                       .map(lambda word: (word, 1))
                       .reduceByKey(lambda x, y: x + y)
                       .collect())
```

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

In [25]:
```
# TEST All together (2d)
Test.assertEquals(sorted(wordCountsCollected), [('cat', 2), ('elephant', 1), ('r
```

```
1 test passed.
```

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 [37]:
```
# TODO: Replace <FILL IN> with appropriate code
uniqueWords = (wordsRDD.distinct().count())
```

```
3
```

In [38]:
```
# TEST Unique words (3a)
```

```
1 test passed.
```

(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.

In [58]:
```
# TODO: Replace <FILL IN> with appropriate code
from operator import add
totalCount = (wordCounts
              .map(lambda (x, y): y)
              .reduce(lambda x, y: x + y))
average = totalCount / float(uniqueWords)
print totalCount
```

```
5
1.67
```

In [57]:
```
# TEST Mean using reduce (3b)
```

```
1 test passed.
```

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 lab. 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 [65]:
```
# 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.
    """
    return (wordListRDD
        .map(lambda word: (word, 1)).reduceByKey(lambda x, y: x + y))
```

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

In [73]:
```
# TEST wordCount function (4a)
Test.assertEquals(sorted(wordCount(wordsRDD).collect()),
                  [('cat', 2), ('elephant', 1), ('rat', 2)],
```

```
1 test passed.
```

(4b) Capitalization and punctuation

Real world files are more complicated than the data we have been using in this lab. 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.

In [118]:
```
# TODO: Replace <FILL IN> with appropriate code
import re
def removePunctuation(text):
    """Removes punctuation, changes to lower case, and strips leading and trail

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

    Args:
        text (str): A string.

    Returns:
        str: The cleaned up string.
    """
    strippedText = re.sub(r'([^\s\w]|_)+', '', text).strip().lower()
    return strippedText
print removePunctuation('Hi, you!')
print removePunctuation(' No under_score!')
```

```
hi you
no underscore
the elephants 4 cats
```

In [119]:
```
# TEST Capitalization and punctuation (4b)
Test.assertEquals(removePunctuation(" The Elephant's 4 cats. "),
                  'the elephants 4 cats',
```

```
1 test passed.
```

(4c) Load a text file

For the next part of this lab, 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 lowercase. Since the file is large we use `take(15)`, so that we only print 15 lines.

In [120]:
```
# Just run this code
import os.path
baseDir = os.path.join('data')
inputPath = os.path.join('cs100', 'lab1', 'shakespeare.txt')
fileName = os.path.join(baseDir, inputPath)

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
```

```
0: 1609
1:
2: the sonnets
3:
4: by william shakespeare
5:
6:
7:
8: 1
9: from fairest creatures we desire increase
10: that thereby beautys rose might never die
11: but as the riper should by time decease
12: his tender heir might bear his memory
13: but thou contracted to thine own bright eyes
14: feedst thy lights flame with selfsubstantial fuel
```

(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.
- The second issue is we need to filter out empty lines.

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.

In [129]:
```
# TODO: Replace <FILL IN> with appropriate code
shakespeareWordsRDD = shakespeareRDD.flatMap(lambda line: line.split(' '))
shakespeareWordCount = shakespeareWordsRDD.count()
print shakespeareWordsRDD.top(5)
```

```
[u'zwaggerd', u'zounds', u'zounds', u'zounds', u'zounds']
927631
```

In [130]:
```
# TEST Words from lines (4d)
# This test allows for leading spaces to be removed either before or after
# punctuation is removed.
Test.assertTrue(shakespeareWordCount == 927631 or shakespeareWordCount == 92890
                'incorrect value for shakespeareWordCount')
Test.assertEquals(shakespeareWordsRDD.top(5),
                  [u'zwaggerd', u'zounds', u'zounds', u'zounds', u'zounds'],
```

```
1 test passed.
1 test passed.
```

(4e) Remove empty elements

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

In [136]:
```
# TODO: Replace <FILL IN> with appropriate code
shakeWordsRDD = shakespeareWordsRDD.filter(lambda word: len(word) > 0)
shakeWordCount = shakeWordsRDD.count()
```

```
882996
```

In [137]:
```
# TEST Remove empty elements (4e)
```

```
1 test passed.
```

(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 lab, 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.

In [141]:
```
# TODO: Replace <FILL IN> with appropriate code
top15WordsAndCounts = wordCount(shakeWordsRDD).takeOrdered(15, lambda (key, val
```

```
the: 27361
and: 26028
i: 20681
to: 19150
of: 17463
a: 14593
you: 13615
my: 12481
in: 10956
that: 10890
is: 9134
not: 8497
with: 7771
me: 7769
it: 7678
```

In [142]:
```
# TEST Count the words (4f)
Test.assertEquals(top15WordsAndCounts,
                  [(u'the', 27361), (u'and', 26028), (u'i', 20681), (u'to', 191
                   (u'a', 14593), (u'you', 13615), (u'my', 12481), (u'in', 1095
                   (u'is', 9134), (u'not', 8497), (u'with', 7771), (u'me', 7769
```

```
1 test passed.
```

In [ ]: