**HU Extension Assignment 04 E63 Big Data Analytics**

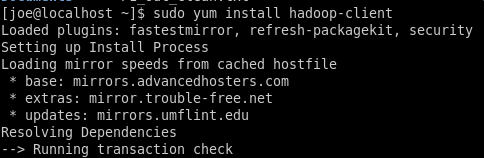
Issued on: February 20, 2016 Due by 11:30PM EST, February 26, 2016

**NOTE: I’m hoping my per-solution text output at the end of this Word document meets related requirements. Please let me know if more/different text capture is necessary for future assignments. At one point I was under the impression you wanted much more actual Command text & non-solution output but that just didn’t make sense.**

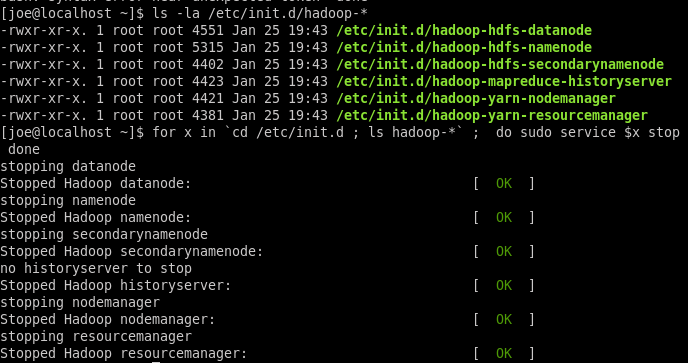
**Problem 1.** Write a working WordCount program using Spark **Java** API that reads a file, e.g. Ulysis/4300.txt from an HDFS directory and writes the results of your calculations to an HDFS file. To improve your word count, remove any punctuation that might have attached itself to your words. Also transform all words into lower case so that the capitalization does not affect the word count. The original code used in lecture notes is provided in the attached mini-example-java.tar file. That archive also contains Maven’s pom.xml file. Run your program and demonstrate that it works. Submit working code inside the customary MS Word Document. Describe steps in your program.

**Solution:**

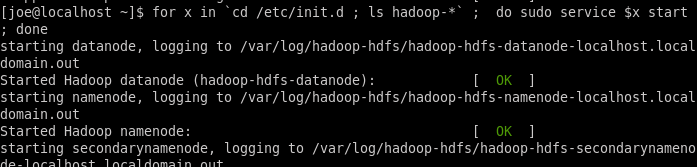
**Spark installation, running through steps from lecture, beginning with installing hadoop-client. Answer yes to all prompts.**



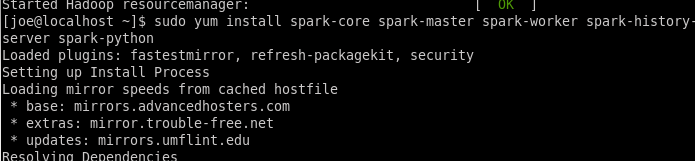
**Following that, confirm HDFS and YARN services are in expected place, stop all that match hadoop-\* filter.**



**And start same, truncated output below.**

**….**

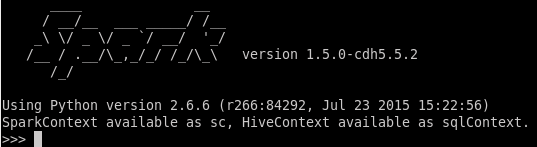
**Now for the actual installation of spark, run the yum w/install command from the lecture pdf. After some minutes it completes.**



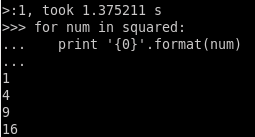
**Going to edit log4j.properties.template as recommended to reduce console output by changing default INFO to highlighted ERROR. (I find it easier to browse & edit via a gpedit session launched as root.)**

|  |
| --- |
| **…** |

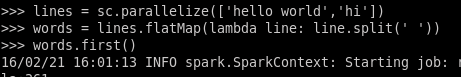
**Don’t see any immediate change to verbosity even w/new Terminal window but pyspark is working. \*\* wasn’t clear from the lecture notes that we needed to create alog4j.properties file using the .template as a template. So I didn’t actually get reduced verbosity until Problem 4.**



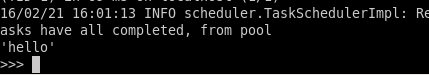
**Use example from lab to make sure it is running correctly, final output is below**



**And the pyspark example, looks good.**

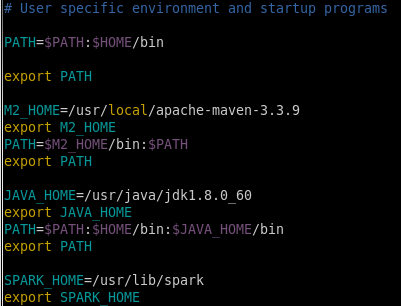


…



**Some Maven setup steps, extract tar.giz to /user/local/ and edit Joe’s .bash\_profile as per pdf. Also set the SPARK\_HOME at this time.**



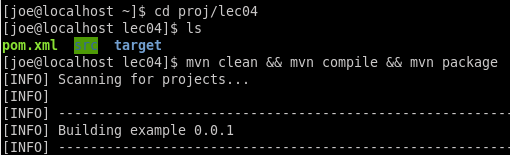


**After source on .bash\_profile to refresh, move to the directory with the pom.xml, on my machine = /home/joe/proj/lec04. Below shows my simplified structure regarding the example project**

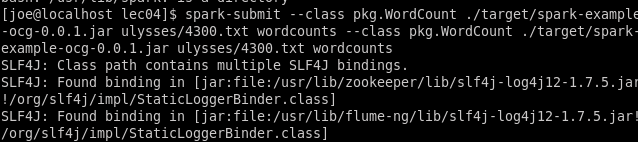
* **pkg is primary package directory, import statements updated to match**
* **pom.xml updated to match, also insert ‘ocg’ as the <name> value**

|  |
| --- |
|  |

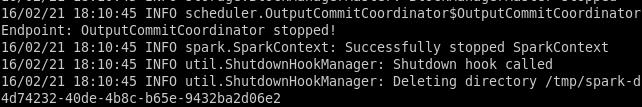
**Run the maven commands from pdf to compile examples.**



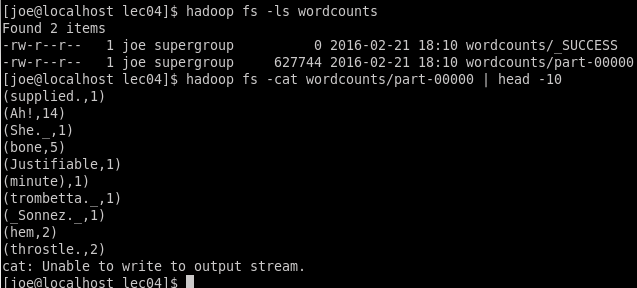
**Build success didn’t mean anything I found out but once I corrected a few configurations, was able to submit the example (non-Eclipse) WordCount class.**



**…**



**Now that it is finished, check the wordcounts output directory and do a head to check on contents of the part file. Looks just like the output displayed in lecture pdf.**



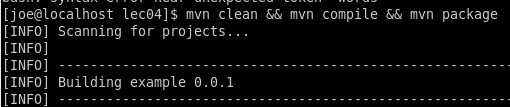
**The example WordCount.java file only needed minor changes to complete the assignment and I’ll only highlight the actual changes, which were all within the call to input.flatMap(). First simply split the words variable, i.e. the line of text, into an array of Strings. Then iterate over the array and within the loop, for each word:**

* **use a regex along with replace() to remove any non-alphanumeric/space characters**
* **then take the cleaned word and use .toLowerCase() to make it all in lower case letters**
* **finally take the result of above and replace the orig ‘dirty’ word in the String array with the clean, lower case version**

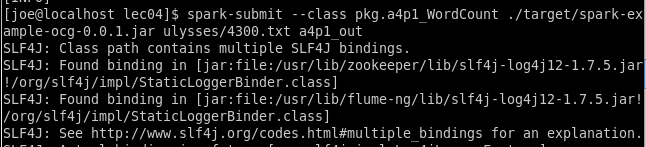
**At the end we have a cleaned array of strings, which can be returned using the .asList casting function that had already been in place.**

|  |
| --- |
| // Load our input data.  JavaRDD<String> input = sc.textFile(inputFile);  // Split up into words.  JavaRDD<String> words = input.flatMap(**new** FlatMapFunction<String, String>() {  **public** Iterable<String> call(String x) {  String[] words = x.split(" ");    **for** (**int** i = 0; i < words.length; i++)  {  //perhaps overly-harsh regex, e.g will remove hyphens from middle of words  String cleanWord = words[i].replaceAll("[^A-Za-z0-9 ]", "");  words[i] = cleanWord.toLowerCase();  }    **return** Arrays.*asList*(words);  }  }); |

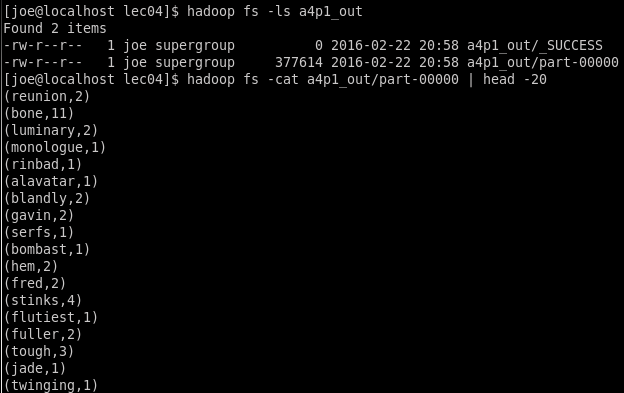
**From the root of my Lec04 directory, which contains my pom.xml, run the Maven commands to compile and build a jar that includes the updated a4p1\_WordCount.java.**



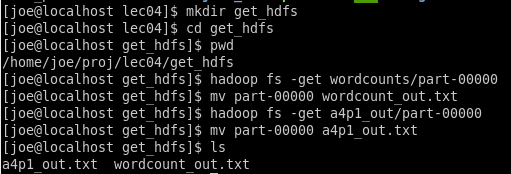
**And then submit the job, passing in my altered class.**



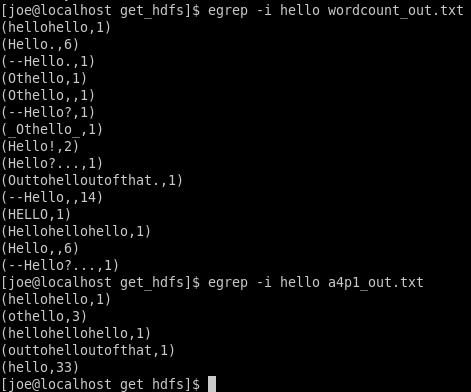
**Check the contents of the a4p1\_out HDFS directory, noting the \_SUCCESS file. Then open the result file and display first 20 lines or so.**



**Everything looks good, no special characters in sight, everything lower case. But should probably try a little harder to confirm output. Create a get\_hdfs directory and get the two files from HDFS, both the original unaltered example WordCount results and then also what my updated class created. Rename both on the local file system.**

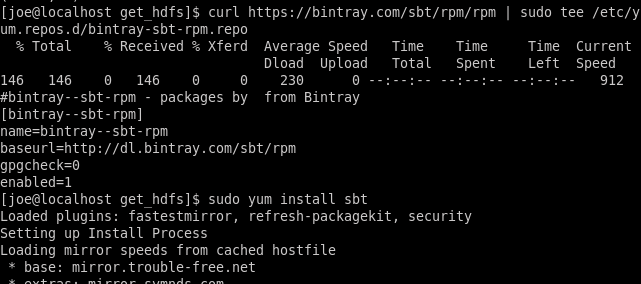


**Now I’ll run the egrep command discussed in lecture to find occurrences of the text “hello” in both the ‘before’ and ‘after’ output files. In the wordcounts file there are something like 20 instances of that text, usually some form of Hello or Othello, with special characters and the like. The same text in my updated a4p1\_out.txt file is found only 5 times, with old Hello version much cleaned up, majority consolidating to simple “hello”, which was found 33 times now. Also note that all of the text is in lower case.**

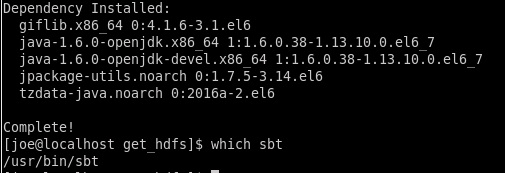


**Problem 2.** Write a working WordCount program using Spark Scala API that reads a file, e.g. Ulysis/4300.txt from a local file system directory and writes the results of your calculations to a local file. To improve your word count, remove any punctuation that might have attached itself to your words. Also transform all words into lower case so that the capitalization does not affect the word count. The original code is provided in the attached mini-example-scala.tar file. That archive also contains Scala Build Tool build.sbt file. Run your program and demonstrate that it works. Submit working code inside the customary MS Word Document. Describe steps in your program.

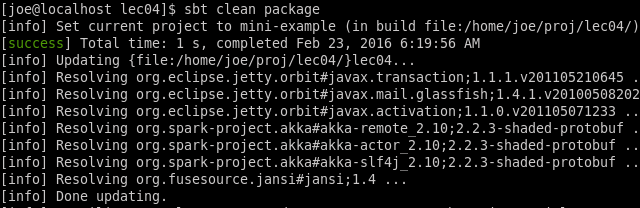
**Solution: Get ready for building the Scala API version of WordCount by installing sbt using the commands discussed in the lecture. When that finishes, run the which command and pass in sbt to confirm success and locate installation directory.**



**…**



**Run the ‘sbt clean package’ command, which takes rather a long time to run the first time. There must have been some installation/initialization because it is quicker the second time.**



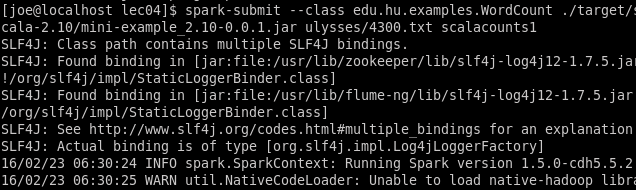
**Use File Browser to confirm output location & presence of .jar.**

|  |
| --- |
|  |

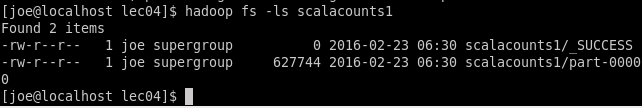
**I’ve hosted the sample .scala files in Eclipse as a convenience only, without any Scala add-on etc.. Additionally, the folder paths have been simplified, partially as an experiment and partially to make things a slight bit easier in terms of navigation.**

|  |
| --- |
|  |

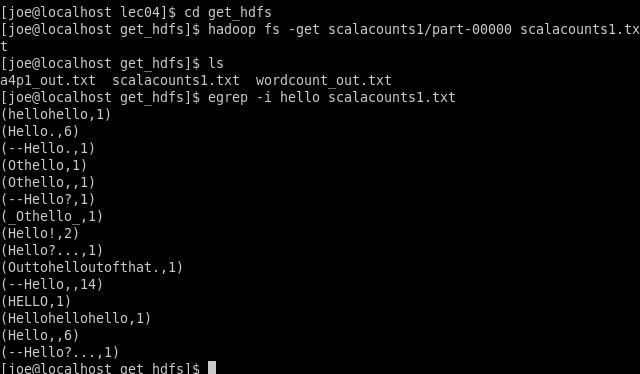
**The original ‘package edu.hu.examples’ remains at the top of the example WordCount.scala though, and apparently the modified directory structure doesn’t have a direct effect on the compilation. I try the original spark-submit command from lecture.**



**An ls passed in to hadoop indicates success.**



**Move to my /get\_hdfs directory and pull out the scala results file, renaming as scala\_counts1.txt. Run the egrep command, looking for “hello” text again and the output looks essentially the same as the Java WordCount example – same text & counts but in a slightly different order. Also notable that kb size on both files is the same.**

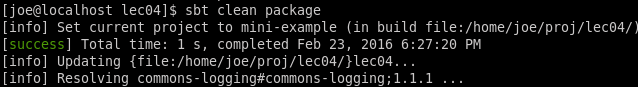


**Now to discuss my needed modifications to the sample WordCount.scala file in order to complete assignment. They are minor in character count but took a while to get right without a proper debug environment. Changes are highlighted in yellow below, all were with the main defined object:**

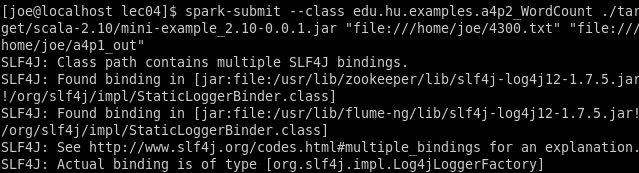
1. **rename object to a4p2\_WordCount**
2. **add a map transformation onto the RDD output of the existing flatMap(), using a lambda within to first run a replaceAll() w/Regex on each word to remove all non-alphanumeric/space characters and then on same word variable force it to lower case with .toLowerCase scala function.**

|  |
| --- |
| object a4p2\_WordCount {  def main(args: Array[String]) {  val inputFile = args(0)  val outputFile = args(1)  val conf = new SparkConf().setAppName("wordCount")  // Create a Scala Spark Context.  val sc = new SparkContext(conf)  // Load our input data.  val input = sc.textFile(inputFile)  // Split up into words.  val words = input.flatMap(line => line.split(" ")  .map(word => word.replaceAll("[^A-Za-z0-9 ]", "").toLowerCase()))    // Transform into word and count.  val counts = words.map(word => (word, 1)).reduceByKey{case (x, y) => x + y}  // Save the word count back out to a text file, causing evaluation.  counts.saveAsTextFile(outputFile)  }  } |

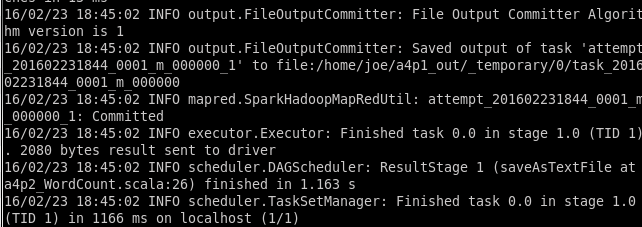
**The step by step for using my new WordCount version begins with compiling by passing ‘clean package’ to sbt.**



**Then I passed in a file path and output path both prefixed with “file:///” so that both the in and the out would be to the local file system. I tried a bit with relative paths but finally went with what worked, which was basing everything out of joe’s home directory, where the 4300.txt file already existed. Output directory name = a4p1\_out. \*\* (oops, should have really been a4p2\_out for sake of consistency but only realized after doing all the work, leaving as-is since this one is to local file system and won’t affect orig problem 1 output in HDFS.)**



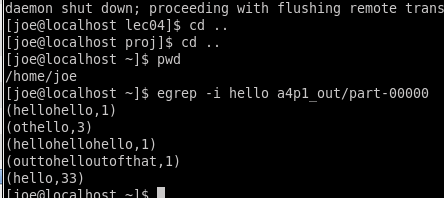
**… (bit below shows the temporary output to …/a4p1\_out directory)**



**To confirm scala code worked as designed, first simply open the output file in gedit since it is on my local file system. Looks good, no special characters, all in lower case. (Text –head output at end of Word document.)**

|  |
| --- |
|  |

**Now for the more exacting test, run the egrep command as was done with java example, looking for the text “hello”. Get the same results, lower-case, no special char etc.**



**Even take a copy of the earlier java output screenshot and compare visually.**

* 

**Problem 3.** Write a working WordCount script using Spark Python API. Read Ulysis (4300.txt) file from an HDFS directory and write the results of your calculations to an HDFS file. To improve your word count, remove any punctuation that might have attached itself to your words. Also transform all words into lower case so that the capitalization does not affect the word count. Run your script using submit-spark tool and demonstrate that it works. Submit working code. Describe steps in your program in the MS Word document.

**Solution: A sample WordCount.py wasn’t supplied but the basic parts were in the lecture notes (though I used single quotes exclusively). Either way it makes sense to display everything since it is so compact. The first items of note are the import statements, the first because I’ll be using regex later, and then the two from pyspark, directly from lecture. Setting up the Spark Configuration and Context objects is straight from the lecture pdf. No pretty code formatting because pyDev wasn’t installed until the next problem.**

|  |
| --- |
| import re  from pyspark import SparkConf, SparkContext    conf = SparkConf().setMaster('local').setAppName('pyWordCount')  sc = SparkContext(conf = conf) |

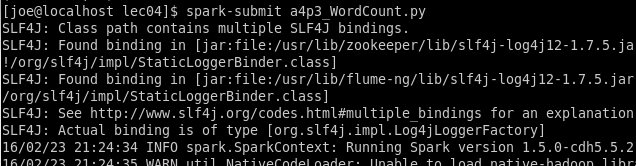
**I load the 4300.txt from the same hdfs directory that it has always been in, apparently the ‘.’ works to indicate root directory. Then I compile a regex object that will remove non-alphanumeric space characters – compiling ahead of time will save a sliver of memory as it loops.**

|  |
| --- |
| lines = sc.textFile('./ulysses/4300.txt')  re\_clean = re.compile(r'[^A-Za-z0-9 ]') |

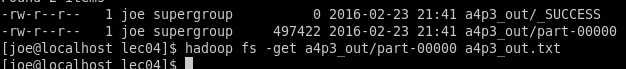
**Then the actual work, again straight from the lecture but broken up into multiple lines. The first RDD uses flatMap() to split the text into a flat series of separate words. The next RDD takes that output and sets a base count = 1 for each word encountered. At this time in the lambda I use the regex .sub() function to replace unwanted special characters and also turn the resulting string into all lower-case. The final RDD receives that output and via reduceByKey groups all of the tuples by the same key and adds their values, i.e. adds together 1 for each same-key. Then simple write the contents of rdd\_summed out to hdfs file system, in a directory = a4p3\_out.**

|  |
| --- |
| rdd\_split = lines.flatMap(lambda line: line.split(' '))  rdd\_count = rdd\_split.map(lambda word: (re.sub(re\_clean, '', word).lower(), 1))  rdd\_summed = rdd\_count.reduceByKey(lambda a, b: a +b)  rdd\_summed.saveAsTextFile('a4p3\_out') |

**Running the code is accomplished by simply passing the .py file on to spark-submit and waiting.**



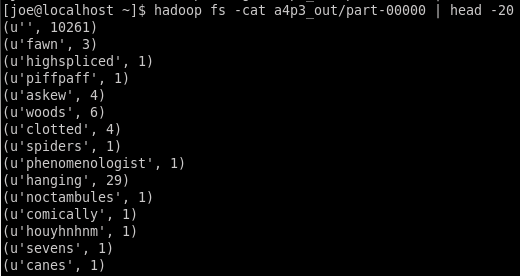
**Check for successful output and pull the file out of hdfs since I’ll want to grep it locally.**



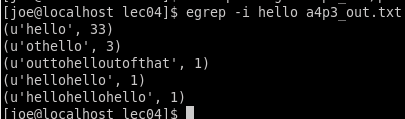
**First do a visual confirm via gedit and results look good. Not great since I’ve wound up with a bunch of empty strings but ‘reasonable’ enough. (I fix that issue in Problem 4 via a .filter() than only outputs a member of the flatMap if it has a len() > 0).**

|  |
| --- |
|  |

**Directly from hdfs.**



**And finally for the real test using egrep to look for the ‘hello’ text. The results are as desired, same lower-case special-character-free words and counts as Java and Scala, albeit in a different order and decorated with Unicode markers.**



**My Java output from Problem 1:**



**Problem 4**. In a Spark API of your choice, write a working BigramCount program which would count occurrences of every pair of consecutive words. You should clean your words just as you did in the previous problem by removing punctuations and cases. However, do not count two words separated by a point at the end of a sentence as a bigram. If you are an experienced programmer add to the bigram count word pairs in which the first word is the last word on the line and the second word is the first word on the subsequent line. If you are not an experienced programmer, than do not do it. Test your program on a small text file, where for comparison, you could identify bigrams manually. Run your program on Ulysis(4300.txt) file and demonstrate that it works. Provide us with the total count of your bigrams, first 20 bigrams and all bigrams containing word “heaven”. Read your file from the local operating system and write results to the local operating system. Include working code in the MS Word Document. Submit the file with the complete working code separately. Describe steps in your program in the MS Word document.

**First I’ll go through the (Python) script itself. Unfortunately, I didn’t have time to even consider looking into the optional exercise of spanning across lines.**

**A few imports regarding regex and actions related to local file system, followed by the standard pyspark imports. The regex/re module is used to pre-compile a regex for cleaning words (removing non-alphanumeric/space characters) and one that will allow multiple sentence delimiters, i.e. period/exclamation point/question mark.**

|  |
| --- |
| import re, os, shutil  from pyspark import SparkConf, SparkContext  re\_clean = re.compile(*r'[^A-Za-z0-9 ]'*)  re\_sentence = re.compile(*r'\. |! |\? '*) |

**Next a clean() function defined, which runs the regex mentioned above and also forces all letters to lower case.**

|  |
| --- |
| def **clean**(word):  return re.sub(re\_clean, *''*, word).lower() |

**Now the function that does most of the non-spark logic. It was originally named func but bigramify is much snazzier. It takes a line of text and splits it into parts using ./!/? as the delimiter. Then each of the sentence parts (though in some cases, if short enough they may be true sentences) is processed, thereby making sure not to count any bigrams that span the sentence delimiter trio. Each sentence part is broken into words, splitting on the space char, which is the default for Python. A list comprehension is used to call clean() on each word and assigned to var named parts. The bigramification actually happens next, where indexing w/in the list comprehension is used to pair each word with the following word, leaving the final word in the sentence part as only a member of the final bigram. Then the items in the resulting list are added on to the bigrams[] created at the beginning of the function. As each sentence part is processed, that bigrams list will grow longer. When the line is fully processed the list of bigrams is returned.**

|  |
| --- |
| def **bigramify**(line):  bigrams = []  sentence\_parts = re.split(re\_sentence, line)  for s in sentence\_parts:  parts = [clean(word) for word in s.split()]  bigrams.extend([(parts[i],parts[i+1]) for i,part in enumerate(parts) if i < len(parts)-1 ])  return bigrams |

**Not too much in the next snippet. I set up a check for the local file system to keep deleting my local output folder before each run. Then the standard Spark configuration and context instanciation. Then we read in the full 4300.txt file from local file system, had also tested with hdfs and of course with a smaller test file.**

|  |
| --- |
| output\_dir = *'/home/joe/bigram\_out'*  if os.path.exists(output\_dir):  shutil.rmtree(output\_dir)  conf = SparkConf().setMaster(*'local'*).setAppName(*'pyBigramCount'*)  sc = SparkContext(conf = conf)  #lines = sc.textFile('./ulysses/4300.txt')  lines = sc.textFile(*'file:///home/joe/4300.txt'*)  #lines = sc.textFile('file:///home/joe/bigram.txt') #test input file |

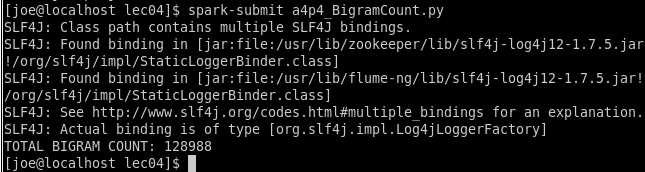
**Then comes a series of RDDs, the first of which runs flatMap, passing the bigramify function discussed earlier. Output that doesn’t have a len() > 0, i.e. is an empty string, is excluded. The other two RDDs basically follow the format of the lecture and Problem 3, doing a map to assign a count = 1 to each initial key (bigram) and then using ReduceByKey to group the keys together and sum up each duplicate key’s 1 value, thereby coming up with a total count of instances for each bigram. The result is saved to a local directly and directly afterward the total bigram count is output to the screen.**

|  |
| --- |
| rdd\_split = lines.flatMap(bigramify).filter(lambda word\_pair: len(word\_pair) > 0)  rdd\_count = rdd\_split.map(lambda word\_pair: (word\_pair, 1))  rdd\_summed = rdd\_count.reduceByKey(lambda a, b: a +b)  rdd\_summed.saveAsTextFile(*'file://'* + output\_dir)  print *'TOTAL BIGRAM COUNT: {0}'*.format(rdd\_summed.count()) |

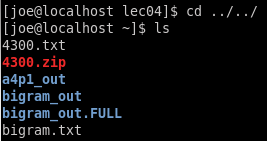
**The final two bits deal with the remaining solution requirements. First one involves using take() to grab the first 20 bigrams present in the rdd\_summed RDD, writing those to a local file. Second is to take that same RDD and run a filter so that only elements meeting a condition are output. In this case that involves checking each member of the bigram to see if its value is exactly equal to text of “heaven”. In order to flush the results of the filter and make them writable to file, call .collect() afterwards.**

|  |
| --- |
| rdd\_twenty = rdd\_summed.take(20)  f = open(output\_dir + *'/twenty.txt'*, *'w'*)  f.writelines([str(bigram) + *'\n'* for bigram in rdd\_twenty])  f.close()  rdd\_heaven = rdd\_summed.filter(lambda result: result[0][0] == *u'heaven'* or result[0][1] == *u'heaven'*).collect()  f = open(output\_dir + *'/heaven.txt'*, *'w'*)  f.writelines([str(bigram) + *'\n'* for bigram in rdd\_heaven])  f.close() |

**Running the script. To begin with we have the 4300.txt file on local system in /home/joe/, as has been show in earlier Problems. Since pathing is coded within my a4p4\_BigramCount.py, all that is need is to pass that file on to spark-submit and it finishes in a few moments. Note that the final line of output includes my final bigram count of 128,988.**



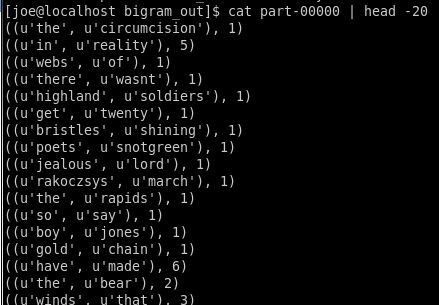
**As per the pathing in the .py, the output should be to a directory named bigram\_out in joe’s home directory and we do see it has been created.**



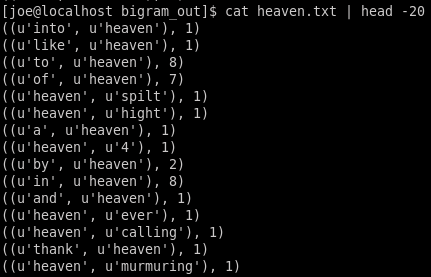
**Move to that dir and do an ls, see that we have expected 4 output files, including a \_SUCCESS.**



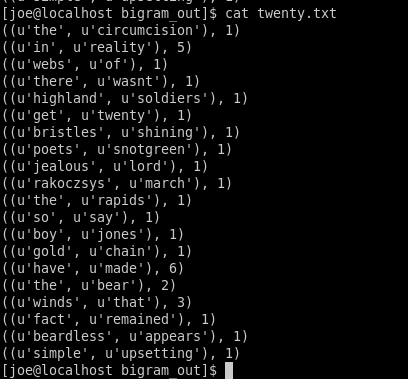
**Check the first several lines of the actual raw spark output via a head command, looks good.**



**Similar for the file holding all of my bigrams that include “heaven”, looks good. I will upload this file as named, heaven, to the Assignment board.**



**And similar for the final file though I can skip the head arg since there should only be twenty lines. Do a visual confirm vs. my screenshot of the full part-00000 file earlier and they do indeed appear to match. This file, twenty.txt, will also go up to the Assignment board.**



Please, describe every step of your work and present all intermediate and final results in a Word document. Please, copy past text version of your command. We cannot retype text that is in JPG images. Please, always submit a copy of original, working scripts and class files you used as separate files. Sometimes we need to run your code and retyping is too costly. Please, submit to the class drop box. For issues and comments visit the class Discussion Board .

**Problem 1**

[joe@localhost lec04]$ **hadoop fs -ls a4p1\_out**

Found 2 items

-rw-r--r-- 1 joe supergroup 0 2016-02-26 06:19 a4p1\_out/\_SUCCESS

-rw-r--r-- 1 joe supergroup 377614 2016-02-26 06:19 a4p1\_out/part-00000

[joe@localhost lec04]$ **hadoop fs -cat a4p1\_out/part-00000 | head -20**

(reunion,2)

(bone,11)

(luminary,2)

(monologue,1)

(rinbad,1)

(alavatar,1)

(blandly,2)

(gavin,2)

(serfs,1)

(bombast,1)

(hem,2)

(fred,2)

(stinks,4)

(flutiest,1)

(fuller,2)

(tough,3)

(jade,1)

(twinging,1)

(jove,5)

(crying,10)

[joe@localhost get\_hdfs]$ **ls**

a4p1\_out1.txt scalacounts1.txt wordcount\_out1.txt

a4p1\_out.txt target wordcount\_out.txt

[joe@localhost get\_hdfs]$ **egrep -i hello wordcount\_out.txt**

(hellohello,1)

(Hello.,6)

(--Hello.,1)

(Othello,1)

(Othello,,1)

(--Hello?,1)

(\_Othello\_,1)

(Hello!,2)

(Hello?...,1)

(Outtohelloutofthat.,1)

(--Hello,,14)

(HELLO,1)

(Hellohellohello,1)

(Hello,,6)

(--Hello?...,1)

[joe@localhost get\_hdfs]$ **egrep -i hello a4p1\_out.txt**

(hellohello,1)

(othello,3)

(hellohellohello,1)

(outtohelloutofthat,1)

(hello,33)

**Problem 2**

[joe@localhost ~]$ **cat a4p1\_out/part-00000 | head -20**

(reunion,2)

(bone,11)

(luminary,2)

(monologue,1)

(rinbad,1)

(alavatar,1)

(blandly,2)

(gavin,2)

(serfs,1)

(bombast,1)

(hem,2)

(fred,2)

(stinks,4)

(flutiest,1)

(fuller,2)

(tough,3)

(jade,1)

(twinging,1)

(jove,5)

(crying,10)

[joe@localhost ~]$ **egrep -i hello a4p1\_out/part-00000**

(hellohello,1)

(othello,3)

(hellohellohello,1)

(outtohelloutofthat,1)

(hello,33)

**Problem 3**

[joe@localhost ~]$ **hadoop fs -cat a4p3\_out/part-00000 | head -20**

(u'', 10261)

(u'fawn', 3)

(u'highspliced', 1)

(u'piffpaff', 1)

(u'askew', 4)

(u'woods', 6)

(u'clotted', 4)

(u'spiders', 1)

(u'phenomenologist', 1)

(u'hanging', 29)

(u'noctambules', 1)

(u'comically', 1)

(u'houyhnhnm', 1)

(u'sevens', 1)

(u'canes', 1)

(u'sprague', 1)

(u'scutter', 1)

(u'originality', 2)

(u'alphabetic', 1)

(u'stipulate', 1)

[joe@localhost ~]$ **cd proj/lec04**

[joe@localhost lec04]$ **egrep -i hello a4p3\_out.txt**

(u'hello', 33)

(u'othello', 3)

(u'outtohelloutofthat', 1)

(u'hellohello', 1)

(u'hellohellohello', 1)

**Problem 4**

(I had already done the code & screenshots before I saw recommended file names)

* Output for “first\_20\_bigrams” is uploaded to site as file named twenty.txt
* Output for “bigrams\_wtih\_heaven” uploaded as heavens.txt.
* Truncated output for the main “all\_bigrams” is below

[joe@localhost bigram\_out]$ **cat part-00000 | head -20**

((u'the', u'circumcision'), 1)

((u'in', u'reality'), 5)

((u'webs', u'of'), 1)

((u'there', u'wasnt'), 1)

((u'highland', u'soldiers'), 1)

((u'get', u'twenty'), 1)

((u'bristles', u'shining'), 1)

((u'poets', u'snotgreen'), 1)

((u'jealous', u'lord'), 1)

((u'rakoczsys', u'march'), 1)

((u'the', u'rapids'), 1)

((u'so', u'say'), 1)

((u'boy', u'jones'), 1)

((u'gold', u'chain'), 1)

((u'have', u'made'), 6)

((u'the', u'bear'), 2)

((u'winds', u'that'), 3)

((u'fact', u'remained'), 1)

((u'beardless', u'appears'), 1)

((u'simple', u'upsetting'), 1)