**Gene Fojtik**

**2/24/2017**

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

Issued on: Feb. 18, 2017 Due on Saturday by 9:30AM EST, Feb. 25, 2017

You can do these problems in any of 4 languages: Python, Scala, Java or R.

**Problem 1.** Go to an online newspaper and select two articles in English on two new movies featured in theaters this spring 2017. Save those articles as .txt files and then import them into two Spark RDD objects, movieA and movieB. Use Spark transformation and action functions to transform those initial RDD-s into RDD-s that contain words and numbers of occurrence of those words in respective article. Eliminate from both lists so called “stop words”. Take the list of stopwords fromthis Web page: <http://www.lextek.com/manuals/onix/stopwords1.html>. List for us 10 most frequent words in each RDD. Subsequently create RDD-s that contain only words unique for each of text. Finally create an RDD that contains only the words common to both texts. Ine latest RDD preserve numbers of occurrences in two articles. In other words a row in your RDD will look like (actor 45 32). List for us a random samples containing 5% of words in the final RDD. We are just practicing RDD transformations and actions. You could implement this problem in a command shell or as a standalone program.

|  |
| --- |
| Step 1: Load and Transform into RDD’s, remove stopwords, List 10 most frequent words in each RDD (movie article)  $ spark-submit problem1.py    Step 2: Create RDD that contains only the common words in articles, preserving the count in both articles, listing a random sample of 5% of the words  $ spark-submit problem1.py    Code:  from pyspark import SparkConf, SparkContext  conf = SparkConf().setMaster("local").setAppName("MyApp")  sc = SparkContext(conf = conf)  sc.setLogLevel("OFF")  movieA = sc.textFile("movie1")  movieB = sc.textFile("movie2")  stopWords = sc.textFile("stopwords")  stopWordsCollected = stopWords.flatMap(lambda x: x.lower().split())  movie1RDD = (movieA  .flatMap(lambda x: x.encode("ascii", "ignore").lower().split())  .map(lambda x: (x,1))  .reduceByKey(lambda x, y: x+ y)  )  movie2RDD = (movieB  .flatMap(lambda x: x.encode("ascii", "ignore").lower().split())  .map(lambda x: (x,1))  .reduceByKey(lambda x, y: x+ y)  )  stopWordsRDD = (stopWords  .flatMap(lambda x: x.split())  .map(lambda x: (x,1))  .reduceByKey(lambda x, y: x+ y)  )  movie1FilterRDD = movie1RDD.subtractByKey(stopWordsRDD)  movie2FilterRDD = movie2RDD.subtractByKey(stopWordsRDD)  movie1Sort = movie1FilterRDD.takeOrdered(10, key = lambda x: -x[1])  movie2Sort = movie2FilterRDD.takeOrdered(10, key = lambda x: -x[1])  #Print results  print ("--------------")  print ("Top words in Movie1")  for (word, count) in movie1Sort:  print("%s: %i" % (word.encode('utf-8'), count))  print ("--------------")  print ("Top words in Movie2")  for (word, count) in movie2Sort:  print("%s: %i" % (word.encode('utf-8'), count))  print ("--------------")  print ("Common words and count")  #RDD of common words in both  commonRDD = movie1FilterRDD.join(movie2FilterRDD)  print ("Common word count %i:" % commonRDD.count())  randomRDD = commonRDD.sample(1,0.04,1234).collect()  for x in randomRDD:  print x  sc.stop() |

**Problem 2**. Consider attached file emps.txt. It contains: name, age and salary of three employees. Create RDD emps by importing that file into Spark. Next create a new RDD emps\_fields by transforming the content of every line in RDD emps into a tuple with three individual elements by splitting the lines on commas. Spark has a class Row and you need to import it in your script or program. Row comes from the same package as class SQLContext. Row class creates rows with named and typed fields. You need to apply “constructor” Row to every tuple in RDD emps\_fields, like:

employees = emps\_fields.map(lambda e: Row(name = e[0], age = int(e[1]), salary = float(e[2])))

e[0], e[1] and e[2] are the first, second and third elements of the tuple e representing a row (line) in RDD emps\_fields. Note that int and float are types of fields in new rows. Newly create RDD employees is now made of Row elements and is ready to be transformed into a DataFrame. You generate a DataFrame by passing an RDD of Row elements to the method createDataFrame() of class SQLContext. Do it. Show the schema of your new data frame. Select complete content of new DataFrame. Transform this DataFrame into a Temporary Table and select names of all employees who have a salary greater than 3500.

Persist your DataFrame as a Parquet file and show that you could exit your pyspark or spark-shell shell, come back into the shell and are able to read the data from the parquet file and recreate the same DataFrame you had originally.

Implement this problem in pyspark of spark-shell or as a standalone program that will be submitted to spark-submit utility.

|  |
| --- |
| **Step A:** Create the RDD from the emps data in hdfs; transform it to a dataframe; show the schema  $ spark-submit prob2\_A.py  Code:  *from pyspark import SparkConf, SparkContext*  *from pyspark.sql import SQLContext, Row*  *conf = SparkConf().setMaster("local").setAppName("MyApp")*  *sc = SparkContext(conf = conf)*  *sqlContext = SQLContext(sc)*  *sc.setLogLevel("OFF")*  *emps = sc.textFile("emps-1")*  *emps\_fields = emps.map(lambda e: e.split(","))*  *employees = emps\_fields.map(lambda e: Row(name = e[0], age = int(e[1]), salary = float(e[2])))*  *employeeDF = sqlContext.createDataFrame(employees)*  *employeeDF.printSchema()*  sc.stop()    **Step B:** Create temp table w/ employee’s salary > 3500, write out to parquet file  $ spark-submit prob2\_B.py  Code:  from pyspark import SparkConf, SparkContext  from pyspark.sql import SQLContext, Row  conf = SparkConf().setMaster("local").setAppName("MyApp")  sc = SparkContext(conf = conf)  sqlContext = SQLContext(sc)  sc.setLogLevel("OFF")  emps = sc.textFile("emps-1")  emps\_fields = emps.map(lambda e: e.split(","))  employees = emps\_fields.map(lambda e: Row(name = e[0], age = int(e[1]), salary = float(e[2])))  employeeDF = sqlContext.createDataFrame(employees)  #Part 2 -> create temp table and write to parquet  #create the temp table  employeeDF.registerTempTable("temp")  #write resultst to parquet file  sqlContext.sql("Select \* from temp where salary > 3500").write.mode("overwrite").save("empsal1", format="parquet")  #delete the temp table  sqlContext.dropTempTable("temp")  Step C: Read parquet file into dataframe, show results  $ submit-spark prob2\_C.py  Code:  from pyspark import SparkConf, SparkContext  from pyspark.sql import SQLContext, Row  conf = SparkConf().setMaster("local").setAppName("MyApp")  sc = SparkContext(conf = conf)  sqlContext = SQLContext(sc)  sc.setLogLevel("OFF")  parquetFile = sqlContext.read.parquet("empsal1")  parquetFile.registerTempTable("temp")  emp = sqlContext.sql("Select \* from temp")  emp.show()  sqlContext.dropTempTable("temp")  sc.stop() |

**Problem 3.** Make sure that two Hive services are up and running. You will find them in the same /etc/init.d directory you were looking for Hadoop services. Click on the browser in your Cloudera QuickStart VM. The welcome screen will open. On the top navigation bar hit Hue. Hue is Cloudera’s browser for Hadoop products. Hue might complain that its configuration is not right. I believe that some user names are missing for some products we do not care right now. On the new navigation bar that will appear select Query Editors and then select Hive. On the left navigation bar you will see a “default” database and several demo tables. In the query window to the right, type “select \* from customers” and then hit the green triangle left of the query window. Next, type “select addresses from customers”. You will see that the customers have shipping and billing addresses.

Hive stores its table as Parquet files in HDFS. Use command:

$ sudo –u hdfs hadoop fs –ls /usr/hive/warehouse

to compare the content of Hive’s warehouse HDFS directory and the table list you see in Hue. Next use the command:

$ sudo –u hdfs Hadoop fs –cat

to expose the actual content of the file stored for the customer table. The content is half readable. That is fine. Capture a portion of it and present in your solution. What type of storage format is used for that file? You can see that “meta information in the file itself.

|  |
| --- |
| Step 1: Run the sql command “Select \* from customers”    Step 2: Run the sql command “Select addresses from customers”    Step 3: List the folder contents of the /user/hive/warehouse directory    Step 4: Cat the contents of the customers file, which is a parquet file.  $ hdfs dfs -cat /user/hive/warehouse/customers/customers |

**Problem 4.** We will explore that customers Hive table in Spark. Follow instructions on slide 62 of lecture notes and create a HiveContext object. Use that object to transfer the data in Hive’s customers table into a Spark DataFrame object. Use that dataFrame to tell us how many shipping addresses there are in each US state. Implement this problem in pyspark of spark-shell or as a standalone program that will be submitted to spark-submit utility.

|  |
| --- |
| Step 1: Use HIVE to query the data from hdfs and place into a dataframe. Use that dateframe to group by shipping state    $ spark-submit hive.py  Code:  *from pyspark import SparkConf, SparkContext*  *from pyspark.sql import SQLContext, HiveContext*  *conf = SparkConf().setMaster("local").setAppName("MyApp")*  *sc = SparkContext(conf = conf)*  *hive = HiveContext(sc)*  *dfs = hive.sql("select addresses from customers")*  *print dfs.groupBy("addresses.shipping.state").count().show()*  *sc.stop()* |

Each problem carries 25% of the grade.

Please, describe every step of your work and present all intermediate and final results in a Word document. Please, copy past text version of all essential command and snippets of results into the Word document. We cannot retype text that is in JPG images. Please, always submit a separate copy of the original, working scripts and/or class files you used as separate files. Sometimes we need to run your code and retyping is too costly. Please include in your MS Word document only relevant portion of the console output or output files. Sometime either console output or the result file is too long and including it into the MS Word document makes that document too hard to read. PLEASE DO NOT EMBED files into your MS Word document. Please, submit to the class drop box. For issues and comments visit the class Discussion Board. You can solve these problems using any language of your choice.