SPARK LAB, FINAL MILESTONE DUE: NOVEMBER 1, 11:59PM EST

1. The "No" List

- No A"I" for code, no A"I" even for comments. A"I" refers to artificial "intelligence" tools like chatgpt, microsoft copilot, etc. You need to know how to write your code and the comments help you figure that out.
- No looking at code from your classmates.
- No stackoverflow or chegg or other sites like that that haven't been put out of business by chatgpt.

2. Preparation

- Code structure video from 10/24. You will need it. Also check out the sparkexample_nocomments folder on github.
- Developing with shell video from 10/24.
- Commenting your code video from 10/24. Also check out the sparkexample_withcomments folder on github. You will be expected to comment your code like this (20 points!!!)
- Review the spark slides. At the very least you should know how to read a file from HDFS and how to create an RDD from a list.

3. Milestones

We will use milestones to make decisions about extensions. Not meeting the milestones decreases the chance of an extension.

- Milestone 1 (by end of class, submit to gradescope just so you have a record of a submission):
  - Either fill in the getTestRDD() and expectedOutput() functions in each of the 4 questions,
  - Or fill in build.sbt, build.properties, and make all of the 4 questions have the recommended structure (see Preparations) section. Do not change the existing function names and types (but add the missing functions that are needed for good structure).
- Milestone 2 (by end of day, submit to gradescope just so you have a record of a submission): do the thing from Milestone 1 you didn't finish.
- Milestone 3 (by final milestone deadline, may be extended depending on earlier milestones): finish the code, test it on your testing RDD, test it on the cluster, submit to gradescope. Code comments: every rdd you create needs a comment that shows what a few lines from it should look like (just like the sparkexample_withcomments sample on github).

4. Questions

Question 0. Autograder is allergic to `._` (used for accessing fields in a tuple) and will not accept any code containing it (actually autograder has no allergies but is a snob and will reject non-stylish code). The **case** trick can help you avoid the dot-underscore and make your code easier to write and understand. Suppose you are writing an anonymous function such as `{(x,y) => x._3 + y._1}` and you know that x is a triple (a,b,c) and y is a tuple (d,e), then you can use the **case** feature of scala to rewrite it more cleanly as: `{case ((a,b,c), (d,e)) => c + d}`, allowing your code to be self-documenting (the structure of x and y becomes clear and you can use meaningful variable names instead of `._1`).

Question 1. In this question, we are going to do a variation of wordcount, where we are only interested in words that have a lowercase "e" (so, we don't want words like "at" to appear in the output, but we do want to know counts of words like "the"). As with the original wordcount program, a word is any sequence of characters that has no space in it (so "that.!" is a word). Write spark code that computes this variation of wordcount and runs it on the War and Peace dataset (**/datasets/wap** in HDFS).

- Save the output in HDFS in the folder **spark1output**
- To split a string on spaces, see the example code mentioned in the Preparations section.
- Update the q1/build.sbt and q1/project/build.properties files as appropriate
- Put your spark code in q1/wordcount_e.scala.
- The bulk of the work should be done by the function **doWordCount** (i.e., it receives an input RDD and produces the answer as its output RDD, but it does not do the saving or reading).
- Check that you are following the milestones carefully (fill in the correct files, test RDD, expected output, the type of comments required by the milestone, etc.).

Question 2. In this question, we are going to do a variation of Question 1, where (in addition to only wanting words with an 'e') we also only care about words that appear 2 or more time total (again we will be using War and Peace).

- Save the output in the folder **spark2output**
- Put your code in q2/wordcount_e2.scala (well structured, yada yada)
- The bulk of the work should be done by the function **doWordCount**.
- Check that you are following the milestones carefully.

Question 3. In this question, we are going to be working with the **retailtab** dataset, located in HDFS in **/datasets/retailtab**. This is a dataset about customer orders from a store. The retailtab dataset is:

- A text file, where each line represents a record. The fields in a record are separated by tabs (represented as `"\t"` in code).
- The dataset has headers which explain what the different fields are
- The field **InvoiceNo** contains the order id (all lines belonging to the same order have the same InvoiceNo).

Your goal is to write spark code to find, for each order (i.e., the InvoiceNo field), the **total cost of the order** (the cost might be negative if the order includes returned items). This means that your output RDD should have type RDD[(String, Double)]. Some things to note:

- Save the output in the folder **spark3output**
- Put your spark code in q3/retail.scala (well strucutred).
- The bulk of the work should be done by the function **doRetail**.
- Check that you are following the milestones carefully.

Question 4. This is a variation of Q3. Your goal is to write spark code to find, for each order, **the total number of items in it and the total cost of the order** (the cost might be negative if the order includes returned items). This means that your output RDD should have type RDD[(String, (Int, Double))]. Some things to note:

- **You can use only 1 wide-dependency operation**
- Save the output in the folder **spark4output**
- Put your spark code in q4/retail2.scala (well strucutred).
- The bulk of the work should be done by the function **doRetail**.
- Check that you are following the milestones carefully.