SI 618 - Homework 4: Map-Reduce in Python Using MrJob

Objectives:

- Get experience with the Map-Reduce computing paradigm.
- Practice how to break down a computing problem into Map-Reduce steps.
- Gain further experience implementing slightly more complex Map-Reduce code in Python using the MRJob module, using local data.
- You will be using the same congressional bills dataset you used for the lab as input in this assignment

Submission Instructions:

After completing this homework, you will turn in a zip file named **youruniquename _si618_hw4.zip** that contains **four** files via Canvas -> Assignments.

1. Your Python script, named **youruniquename** _si618 _hw4_part1.py
2. Your output file, named **youruniquename**_si618 _hw4_output_part1.txt. **This should be the output produced by your Python script.**
3. Your Python script, named **youruniquename** _si618 _hw4_part2.py
4. Your output file, named **youruniquename**_si618 _hw4_output_part2.txt. **This should be the output produced by your Python script.**

This homework involves multiple steps. You should review all steps before starting, to think about how you're going to structure your code that can be used for both the earlier and later steps. You should then complete the steps one by one and verify that you've completed each correctly before moving on to the next one.

MRJob Documentation: https://pythonhosted.org/mrjob/

A quick intro to Object Oriented Programming:
https://jeffknupp.com/blog/2014/06/18/improve-your-python-python-classes-and-object-oriented-programming/

Step 1: Create a new Python file
This time, we will not be providing a template. Instead we will start with a blank file and build the program step by step. Create a new Python file and save it as **youruniquename**_si618_ hw4_part1.py

Step 2: Create an MRJob subclass
To create subclass (derived class) of MRJob, we first need to import the MRJob class into our file and then create our new class that inherits from MRJob. We can do this using:

```
from mrjob.job import MRJob

class MRMostUsedWords(MRJob):
  pass

if __name__ == "__main__":
  MRMostUsedWords.run()
```

We also need to call the 'run' method of the MRJob subclass MRMostUsedWords.

Step 3: Try running this file
python **youruniquename**_si618_hw4_part1.py ./bills -o ./output

You will see an error like:
ValueError: Step has no mappers and no reducers

This error means that MrJob was imported and called successfully, but it is expecting you to define at least one mapper or reducer so that something is actually computed for the input file.

Step 4: Write a mapper that splits a line of input into words
Similar to what we did in the lab, we define a **mapper** function that takes a line from the input file as an input, splits the bill title into words and yields a tuple in the form (YEAR<tab>WORD, COUNT_FOR_THIS_WORD). *Remember, a mapper or reducer always return one or more (key, value)* tuples.

```
class MRMostUsedWords(MRJob):
  OUTPUT_PROTOCOL = mrjob.protocol.TextProtocol
  def mapper(self, _, line):
    # your code goes here
```

We use _ as the second parameter as a convention, because we want to ignore any value that is passed as that parameter.

Use a regular expression to find **all long words (words with at least 4 letters)** in the line. Make sure you convert them to lowercase.

The mapper should produce, using the yield keyword, one tuple in the form (YEAR <tab>WORD, COUNT_FOR_THIS_WORD) for each matching word in the bill title.

Congratulations, you just wrote the Mapping stage of the MapReduce flow.

Step 5: Write a combiner that sums up the count for each year, word pair
The shuffling stage re-distributes the mapping values by keyword so that they can be passed to the reducer to get a result. We want to minimize the network data transfer among nodes at the shuffling stage. So, we introduce the **combiner.** The combiner can consist of any operations that

can minimize the data transfer at the shuffling stage. In our case, we can combine the counts of year, word pairs at each node, so that we don't send the same pair multiple times across the network.

The combiner function is similar to a mapper and a reducer, such that it also yields one or more (key, value) pairs.

In this combiner step, we sum up the counts of all different words at each node. Now your code should look something like this:

```
class MRMostUsedWords(MRJob):
  OUTPUT_PROTOCOL = mrjob.protocol.TextProtocol
  def mapper(self, _, line):
    # your code goes here

  def combiner(self, word, counts):
    # your code goes here
```

The combiner method should produce, using the yield keyword, a tuple of the form (YEAR <tab>WORD, TOTAL_COUNT_FOR_WORD).

Step 6: Write a reducer that sums up the count for each word at the end
At this stage, the whole file has been read and we can now find out the final total count for each word. Your code should look something like this:

```
class MRMostUsedWords(MRJob):
  OUTPUT_PROTOCOL = mrjob.protocol.TextProtocol
  def mapper(self, _, line):
    # your code goes here

  def combiner(self, word, counts):
    # your code goes here

  def reducer(self, word, counts):
    # your code goes here
```

In this case, the arguments and the output of both combiner and reducer methods look the same, because they are essentially trying to do the same thing, but the combiner method is executed at an earlier stage than the reducer and both are usually executed at different computing nodes. Also, the reducer gets the final list of counts for each (year, word) and not an intermediate list like the combiner does.

Run your code:
Python3 **youruniquename** si618_hw4_part1.py ./bills -o ./output
(**Note that, as with the lab, you will need wrap your mapper code in a try-except block to prevent malformed lines in the input file from breaking your code)**

The output files /**part-*** under the output folder will contain the (year<tab>word<tab>frequency) triples. Consolidate them under **youruniquename** _si618_hw4_output_part1.txt as follows:
cat ./output/part* > **youruniquename** _si618_hw4_output_part***1***.txt

Your output should look like si618_hw4_part1_desired_output.txt (included in the homework zip).

Step 7: Find out the most frequent word for each length:
Now copy **youruniquename**_si618_hw4_part1.py to **youruniquename**_si618_hw4_part2.py. You need to change your code now to identify the most frequent words of each length at least 4 letters long **irrespective of the year of the bill**. The output of the reducer needs to be passed to the next step in the pipeline, which will find out the most frequently used word of each length. This cannot be an independent step that is done in parallel, since selecting the maximum count is a comparison operation that requires counts for all words of a given length to be finished. So, the reducer should yield only values to be passed to the next step, i.e. it should yield a tuple in the form (WORD_LENGTH, TUPLE_OF_WORD_AND_TOTAL_COUNT). This will combine all the (WORD, TOTAL_COUNT_FOR_WORD) tuples for each word length as a list of tuples (technically a generator/lazy list of tuples) under the same key, thus making it available to a single node for processing the most frequent word in the next step.

As an example, typically we would yield (WORD, TOTAL_COUNT_FOR_WORD) or (TOTAL_COUNT_FOR_WORD, WORD), which would have resulted in an output like for words of length 5:
(“apple”, 1)
(“mango”, 2)
(“peach”, 3)
…

The above output yields multiple key, value tuples. In contrast to this, we want to pass such a tuple as a VALUE to the next step for processing, such that the key is 5 and the values get accumulated, i.e, we yield:
(5, (“apple”, 1))
(5, (“mango”, 2))
(5, (“peach”, 3))

Step 8: Making our own MRSteps
So far, we have a mapper, combiner and reducer. A mapper pre-processes the data for further processing by a reducer. A combiner aggregates the data to limit the amount of copying between the different nodes. A reducer processes this data resulting in one or more key-value pairs (which is a tuple, not to be confused with a dictionary).

For the last step, finding the most frequent word across for each word length, we need an additional reducer that takes the data from the previous reducer step and outputs the most frequent word for each length. (A mapper isn’t required because there is no pre-processing.) But, using the default configuration of MRJob, we cannot have a second method also named ‘reducer’. To overcome this limitation, MRJob allows us to define additional custom

computation steps using the MRStep class that allows us to write our own sequence of steps (defined using multiple MRStep objects) that MRJob should follow.

Here's an example of how to define two MRStep objects that perform the initial map/combine/reduce in the first MRStep, and the final reduce operation in the second MRStep. Your code should look like this:

```
from mrjob.job import MRJob
from mrjob.step import MRStep
import re

class MRMostUsedWords(MRJob):
  OUTPUT_PROTOCOL = mrjob.protocol.TextProtocol
  def mapper_get_words(self, _, line):
    # your code goes here

  def combiner_count_words(self, word, counts):
    # your code goes here

  def reducer_count_words(self, word, counts):
    # your code goes here

  def reducer_find_max_word(self, length, word_count_pairs):
    # your code goes here

  def steps(self):
    return [
      MRStep(mapper=...,
          combiner=...,
          reducer=...),
      MRStep(reducer=...)
    ]
```

Fill in the "…". Using the step method of MRJob, we can return a list of MRStep objects and thus control the flow of execution. The first MRStep uses a mapper, combiner and reducer, while the second MRStep uses a reducer that uses the data passed by the previous reducer to find the most frequent word.

Step 9: Find the most frequent word of a certain length

Write the code for reducer_find_max_word such that it yields a tuple (LENGTH, WORD, TOTAL_COUNT) or (LENGTH, TOTAL_COUNT, WORD) for the most frequent word for each length. The output may vary depending on whether you decide to use sort or max.

max function uses the first value in the tuple for comparison, whereas you can sort it and return the one tuple that represents the most frequent word.

Run this code:

Python3 **youruniquename**_si618_hw4_part2.py ./bills -o ./output

The output files **part-00000*** should contain one line for each length:

Step 9: Combine and rename output file
Combine the output files into a single file **youruniquename**_si618_hw4_output_part2.txt similar to what you did for part 1 for submission.

Ungraded Optional Challenge: Create a new version of the si618_hw4_part1.py (name it si618_hw4_part3.py). Apply what you learned with the writing si618_hw4_part1.py to add a new reducer that would output the word frequencies within year sorted from most frequent to least frequent.

Rubric:
Step 4 (writing the correct mapper): 20 pt
Step 5 (writing the correct combiner): 20 pt
Step 6 (writing the correct reducer for part-1): 20 pt
Step 7 (editing the mapper correctly and writing the correct reducer for the second step in part-2): 20pt
Step 8 (writing the correct step function): 20 pt