SI 618 - Homework 5: 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_hw5.zip** that contains **four** files via Canvas -> Assignments.

1. Your Python script, named **youruniquename** _si618 _hw5_part1.py
2. Your output file, named **youruniquename**_si618 _hw5_output_part1.txt. **This should be the output produced by your Python script.**
3. Your Python script, named **youruniquename** _si618 _hw5_part2.py
4. Your output file, named **youruniquename**_si618 _hw5_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://mrjob.readthedocs.io/en/latest/

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

You are given a dataset of bills introduced at the US congress from the 80th congress up to the 116th (partial) in the zip file. The dataset consists of 10 files in the tsv (tab-separated values) format. Each row of a file contains the following 9 fields

1. Id – unique identifier of the bill
2. Type of bill {"HR": house bill, "s": senate bill, "sres": senate resolution, "hcon": house concurrent resolution, "scon": senate concurrent resolution, "hjres": house joint resolution, "sjres": senate joint resolution}
3. Title of the resolution
4. Which chamber introduced the bill (0 – House, 1 – Senate)

5. The congress in which the bill was introduced (80 to 116)
6. The year the bill was introduced
7. Did the bill pass in the House? (Boolean)
8. Did the bill pass in the Senate? (Boolean)
9. Was the bill passed into law? (Boolean)

Credit: This dataset is a formatted version derived from the source datasets at http://www.congressionalbills.org/ (Adler and Wilkerson)

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_hw5_part1.py

Step 2: Create an MRJob subclass

To create a 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 MRMostUsedWordStems(MRJob):
  pass

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

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

Step 3: Try running this file

python **youruniquename**_si618_hw5_part1.py ./bills -o ./output

You will see this error or a similar one:

ValueError: Job has no steps!

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 stemmed words

Stemming is the process of removing the morphological affixes and reducing a word to its word stem. For example, the words "program", "programs", "programming", "programmer", and "programmers" could all be reduced to the stem "program". Stemming is an important technique in natural language processing (NLP).

Different from the lab, we want to define a **mapper** function that takes a line from the input file as an input, **filters the long words**, **stems the words**, and yields a tuple in the form (CONGRESS<tab>WORD_STEM, COUNT_FOR_THIS_ WORD_STEM), where CONGRESS is the congress in which the bill was introduced (it should be a number between 80 and 116).

To filter the words, please use a regular expression to find **all long words (words with at least 4 letters)** in the line.

Next, to stem these filtered words, we will utilize the **nltk** (Natural Language Toolkit) package. You will need to install this package, and then import the **PorterStemmer** from this package. The documentation for how to use the stemmer can be found at https://www.nltk.org/howto/stem.html.

We won't dig into the implementation of the stemmer here, but if you are interested, the source code of the PorterStemmer could be found at https://docs.huihoo.com/nltk/0.9.5/api/nltk.stem.porter.PorterStemmer-class.html)

The template of the mapper is here:

```
class MRMostUsedWordStems(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.

As we said before, the mapper should produce, using the yield keyword, one tuple in the form (CONGRESS<tab>WORD_STEM, COUNT_FOR_THIS_ WORD_STEM) for each long word in the entire line. *Remember, a mapper or reducer always returns one or more (key, value)* tuples.

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

* Since stemming the words will cost extra time, you might want to **test your code on a smaller input dataset first (e.g., 2 bill files)** instead of running the code on 9 bill files directly. After you confirm that your code is working properly, then use all the bill files to generate the result. It should take you around 3 minutes to run the code once on all the bill files.

Step 5: Write a combiner that sums up the count for each congress, stem 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 (congress, word stem) 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 stems 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_stem, counts):
    # your code goes here
```

The combiner method should produce, using the yield keyword, a tuple of the form (CONGRESS<tab>WORD_STEM, COUNT_FOR_THIS_ WORD_STEM).

Step 6: Write a reducer that sums up the count for each word stem 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 stem. Your code should look something like this:

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

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

  def reducer(self, word_stem, 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 (congress, word_stem) and not an intermediate list as the combiner does.

* OUTPUT_PROTOCOL = mrjob.protocol.TextProtocol will restrict the format of the output, so you will need to cast the type in your last reducer.

Run your code:
python **youruniquename** si618_hw5_part1.py ./bills -o ./output
(**Note that, as with the lab, you will need to 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 (congress<tab>word_stem<tab>frequency) triples. Consolidate them under **youruniquename** _si618_hw5_output_part1.txt as follows:

cat ./output/part* > **youruniquename** _si618_hw5_output_part1.txt

Your output should look like si618_hw5_part1_desired_output.txt (included in the homework zip). Note that the order of the lines might be different, but the content should be similar.

Step 7: Find out the most frequent word stem for each length:
Now copy **youruniquename**_si618_hw5_part1.py to **youruniquename**_si618_hw5_part2.py. You need to change your code now to identify the most frequent word stems of words that are at least 4 letters long **irrespective of the congress in which the bill was introduced**. 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 stem of each length **(note that this is the length of the word stem, not the length of the original word).** 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 word stems 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_STEM_LENGTH, TUPLE_OF_WORD_STEM_AND_TOTAL_COUNT). This will combine all the (WORD_STEM, TOTAL_COUNT_FOR_WORD_STEM) tuples for each word stem 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 stem in the next step.

As an example, typically we would yield (WORD_STEM, TOTAL_COUNT_FOR_WORD_STEM) or (TOTAL_COUNT_FOR_WORD_STEM, WORD_STEM), which would have resulted in an output like for word stems of length 5:
(“sleep”, 1)
(“think”, 2)
(“laugh”, 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, (“sleep”, 1))
(5, (“think”, 2))
(5, (“laugh”, 3))

Step 8: Making your 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 stem for each word stem length, we need an additional reducer that takes the data from the previous reducer step and outputs the most frequent word stem 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 your 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 MRMostUsedWordStems(MRJob):
  OUTPUT_PROTOCOL = mrjob.protocol.TextProtocol
  def mapper_get_word_stems(self, _, line):
    # your code goes here

  def combiner_count_word_stems(self, word_stem, counts):
    # your code goes here

  def reducer_count_word_stems(self, word_stem, counts):
    # your code goes here

  def reducer_find_max_word_stem(self, length, word_stem_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 stem.

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

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

The 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 stem.

Run this code:

python **youruniquename**_si618_hw5_part2.py ./bills -o ./output

* Again, you might want to **test your code on a smaller input dataset first (e.g., 2 bill files)** instead of running the code on 9 bill files directly. After you confirm that your code is working properly, then use all the bill files to generate the result. It should take you around 4 minutes to run the code once on all the bill files.

The output files **part-00000*** should contain one line for each length. Please the output files into a single file **youruniquename_**si618_hw5_output_part2.txt similar to what you did for part 1 for submission. Your output should look like si618_hw5_part2_desired_output.txt (included in the homework zip). Note that the order of the lines might be different, but the content should be similar.

Ungraded Optional Challenge: Create a new version of the si618_hw5_part1.py (name it si618_hw5_part3.py). Apply what you learned with the writing si618_hw5_part1.py to add a new reducer that would output the word stem frequencies for each congress.

Rubric:
Step 4 (writing the correct mapper): 20 pt
Step 5 (writing the correct combiner): 10 pt
Step 6 (writing the correct reducer for part 1): 10 pt
Step 7 (writing the first step for part 2 correctly): 20pt
Step 8 (writing the correct step function): 20 pt
Step 9 (writing the second step for part 2 correctly): 20pt