Assignment 1: Algorithms in MapReduce

(Adapted from Bill Howe's programming assignment, Coursera course: Introduction to Data Science)

In this assignment, you will be designing and implementing MapReduce algorithms for a variety of common data processing tasks.

The goal of this assignment is to give you experience "thinking in MapReduce." We will be using small datasets that you can inspect directly to determine the correctness of your results and to internalize how MapReduce works.

Write your own code!

For this assignment to be an effective learning experience, you must write your own code! I emphasize this point because you will be able to find Python implementations of most or perhaps even all of the required functions on the web. Please do not look for or at any such code! **Do no share code with other students in the class!!**

Here's why:

- The most obvious reason is that it will be a huge temptation to cheat: if you include code written by anyone else in your solution to the assignment, you will be cheating. As mentioned in the syllabus, this is a very serious offense, and may lead to you failing the class.
- However, even if you do not directly include any code you look at in your solution, it surely will influence your coding. Put another way, it will short-circuit the process of you figuring out how to solve the problem, and will thus decrease how much you learn.

So, just don't look on the web for any code relevant to these problems. Don't do it.

Python MapReduce Framework

You will be provided with a python library called MapReduce.py that implements the MapReduce programming model. The framework faithfully implements the MapReduce programming model, but it executes entirely on a single machine -- it does not involve parallel computation.

Note that we have added a new import ("import re" for regular expression support) to the wordcount.py. Please use the one provided here as the template, instead of the one posted earlier for tutorial.

Here is the word count example discussed in class implemented as a MapReduce program using the framework:

```
# Part 1
mr = MapReduce.MapReduce()

# Part 2
def mapper(record):
    # key: document identifier
    # value: document contents
    key = record[0]
    value = record[1]
    words = value.split()
    for w in words:
      mr.emit_intermediate(w, 1)
```

```
# Part 3
def reducer(key, list_of_values):
    # key: word
    # value: list of occurrence counts
    total = 0
    for v in list_of_values:
      total += v
    mr.emit((key, total))

# Part 4
inputdata = open(sys.argv[1])
mr.execute(inputdata, mapper, reducer)
```

In Part 1, we create a MapReduce object that is used to pass data between the map function and the reduce function; you won't need to use this object directly.

In Part 2, the mapper function tokenizes each document and emits a set of key-value pairs. The key is a word formatted as a string and the value is the integer 1 to indicate an occurrence of word.

In Part 3, the reducer function sums up the list of occurrence counts and emits a count for word. Since the mapper function emits the integer 1 for each word, each element in the list_of_values is the integer 1.

The list of occurrence counts is summed and a (word, total) tuple is emitted where word is a string and total is an integer.

In Part 4, the code loads the JSON file and executes the MapReduce program which prints the result to stdout.

Submission Details

For each problem, you will turn in a python script (except for Problem 3, which requires the submission of two python scripts), similar to wordcount.py, that solves the problem using the supplied MapReduce framework.

When testing, make sure MapReduce.py is in the same directory as the solution script.

To allow you to test your programs, sample data will be provided in the data folder, and the corresponding solutions will be provided for each problem in the solutions folder.

Your python submission scripts are required to have a mapper function that accepts at least 1 argument and a reducer function that accepts at least 2 arguments. Your submission is also required to have a global variable named mr which points to a MapReduce object.

If you solve the problems by simply replacing the mapper and reducer functions in wordcount.py, then these conditions will be satisfied automatically.

What you will turn in these **FOUR** programs (with five python scripts to be submitted):

1. Program that computes tf (term frequency) and df (document frequency) of distinct words in a given set of documents, called tf_df.py. See the details below for the explanation of tf and df.
2. Program that finds frequent 2-itemsets in a given set of transactions, called `itemsets.py`. See below for the details on the definition of transactions and frequent k-itemsets.

3. Program that computes the square of a given square matrix using the one-phase approach, called squared_one.py.
4. Program that computes the square of a given square matrix using the two-phase approach, called squared_two_1.py (for the first phase) and squared_two_2.py (for the second phase).

Finally, to facilitate our grading of your submission, we require that:

1. **Your output follow exactly the format as given in the solution files.**
2. **Add your first name and last name to the beginning of your python scripts, e.g., john_smith_tf_df.py.**

Problem 1 (tf-df)

Background: Given a document, ***tf*** (term frequency) of a term in the document is the number of occurrences of the term in the document. For example, consider the following short document D:

*In a swanky hotel ballroom, he told the crowd of alumni and donors that Texas had become the largest feeder of **students** to USC after California, and that **students** from their state scored significantly higher on the SAT than the average of all applicants.*

The term "students" occurs twice in the documents, so its tf is 2 for this document.

In this problem, we use term to refer to token (a sequence of characters) in the document that contains only alphanumeric and underscore characters. We ask you to normalize the tokens, so that two tokens which differ only in the case of their characters will be considered the same token. For example, "students" is the same as "Students". (Note that we will not perform other normalization, e.g., stemming or removing the suffix of words such as turning "students" into "student", nor remove stopwords such as "the", "an").

Moreover, given a collection of documents, ***df*** (document frequency)of a term refers to the number of documents in the collection where the term occurs. For example, suppose that there are 10 documents in the collection and that "students" occur in three of them, e.g., one of which may be the document D above. Then the document frequency of "students" is 3.

Tf and df of a term indicate the importance of the term in the document and across the documents. They are important statistics for effective information retrieval. You may refer to Section 1.3.1 for more details. (Note that df can be used to compute inverse document frequency or IDF)

Task: Write a MapReduce program, tf_df.py, that finds unique tokens in a given collection of documents, and for each token, computes its term frequency in each documents, and also its document frequency.

Mapper Input

The input is a 2 element list: `[document_id, text]`, where `document_id` is a string representing a document identifier and `text` is a string representing the text of the document. As described above, a token in the document is a sequence of alphanumeric characters or underscores. Note that it is not sufficient to simply split the content into tokens using value.split() (which splits by white spaces). You may need to use the search function with regular expression support, e.g., re.findall(). Remember also to lowercase the original tokens to turn them into terms.

Reducer Output

The output should be a set of <term, df, a list of <document_id, tf>> tuples. Note that the framework turns your output tuples into JSON arrays.. For example,

["nearly", 2, [["austen-emma.txt", 2], ["chesterton-ball.txt", 3]]]

Which says the term “nearly”, has a document frequency 2. It appears in document “austen-emma.txt” twice, and “chesterton-ball.txt” three times.

You can test your solution to this problem using the data file books.json:

python tf_df.py books.json

You can verify your solution against tf_df_output.json.

Problem 2 (frequent itemsets)

In this problem, you are given a set of transactions. Each transaction contains a set of items purchased by some customer. Items are represented as integers. For example, [1, 3, 5] is a transaction that contains items 1, 3, and 5. You may assume that item numbers are sorted in the ascending order.

We consider the problem of finding itemsets which frequently occur in the transactions. The problem is an important part of the association rule mining which we will see in the coming lectures. An itemset is a set of items and a k-itemset is an itemset that contains k items. For example, the transaction [1,3,5] contains three 2-itemsets [1,3], [1,5], and [3,5]. A frequent itemset is one that occurs in at least x number of transactions, where x is called minimum support count of the itemset.

In this problem, you are asked to find all 2-itemsets that appear in a given set of transactions at least 100 times.

Map Input

Each input record is a transaction that contains a set of items purchased in that transaction. Remember as mentioned above, the items are sorted by their item numbers.

Reduce Output

The output for each reducer should be a 2-itemset, e.g., [38, 39], that occurs in at least 100 transactions.

You can test your solution to this problem using the data file `transactions`.json:

```
$ python itemsets.py transactions.json
```

You can verify your solution by comparing your result with the file itemsets_output.json.

Problem 3 (squaring matrix, one-phase)

In this problem, you are provided with a matrix, say A, and asked to write a MapReduce program to compute its square (A^2).

The matrix is given in a sparse matrix format, where elements in the matrix are stored individually as a tuple <i, j, A[i,j]>. For example, [0, 0, 63] means that A[0,0] = 63

Recall that we saw two methods of multiplying matrices in class. One that uses only one MapReduce phase; the other using two phases.

In this problem, we ask you to implement a MapReduce program, squared_one.py, using the one-phase approach, that is, the reducer will be responsible for both multiplication of corresponding elements (both from the same matrix in this case) and addition of products.

Map Input

Each mapper will take as the input a tuple (i.e., a Python list) that represents a matrix element.

Reduce Output

The output should be a set of tuples that represent the result matrix in the similar sparse format.

You can test your solution to this problem using the sample input, matrix.json.

```
$ python squared_one.py matrix.json
```

You can verify your solution by comparing your result with the file squared_output.json.

Problem 4 (squaring matrix, two-phase)

This problem is similar to Problem 3, but you are asked to write a two-phase program. Recall that in the first phase, multiplication results of corresponding elements from the matrices are obtained. Then, in the second phase, individual products are added together to produce the final element in the output (squared) matrix.

Name the MapReduce program for the first phase as squared_two_1.py, and the second phase as squared_two_2.py.

Map Input (phase one)

Same as Problem 3.

Reduce Output (phase one)

Remember the output from the reducers will be given as the input to the next phase of MapReduce. Refer to squared_two_1_output.json for sample output from phase one.

Map Input (phase two)

Take output from phase 1 as the input. Each mapper will get a tuple from the file, e.g., [i, k, v], representing a value v (product) to be added (with other products) to give the value for the element [i, k] in the output.

Reduce Output (phase two)

Same as Problem 3.