Assignment 3 « MapReduce »

Brief

- Due date: 11:59pm 03/05/2020
- Stencil: `cs1951a_install mapreduce`
- Data: `/course/cs1951a/pub/mapreduce/data/` (will be installed automatically in `~/course/cs1951a/data/mapreduce`)
- Handin: `cs1951a_handin mapreduce`
- Files to submit: README, similarities.py, inverted_index.py, join.py, written_questions.pdf

Overview

O[[WZ!JVSHIYLZLHYJONVVNSLJVTKYP]LKÅ?0K:>2^QM)Q

In this assignment, you will be designing and implementing MapReduce algorithms for a variety of common data processing tasks. You still will be performing most of the steps on your machine or in your `Google Cloud Shell`. For the huge dataset, you can submit the job to a Dataproc cluster we prepared for you in Google Cloud Platform (GCP).

In part 1 of this assignment, you will solve two simple problems on small datasets. You will build the MapReduce pipelines and implement your mappers and reducers.

In part 2 of this assignment, you will implement a movie recommendation system. Part of the MapReduce pipeline is provided. You will design the remaining part. And you will also need to implement the mappers and reducers. There are three datasets in part 2: small, big, and huge. For the small and big dataset, you can directly run your program in your Google Cloud Shell. For the huge dataset, you will have to submit your job to our Dataproc cluster.

How to Start

Option 1: Google Cloud Shell

Set up virtual environment: You need to create a Python virtual environment and install `PySpark` in your Google Cloud Shell. If you haven't set up Google Cloud Shell before, please look at Google Cloud Shell Setup (https://colab.research.google.com/drive/1dfIXIdSWKwjfBj4pPCdfuhDN1koJWYFL).

Write Code: You can complete the assignment on your own laptop and upload (https://colab.research.google.com/drive/1dfIXIdSWKwjfBj4pPCdfuhDN1koJWYFL#scrollTo=t5K1oXxK9VJC) the files to your Google Cloud Shell. Or you can use command-line text editor such as `vim` and `nano` to edit your code in Google Cloud Shell.

Execute Code: Activate the virtual environment and run your program from the command line. The virtual env can be activated by typing `source ~/venv/bin/activate`.

Option 2: Department Machine

Write Code: Use whatever editer or IDE you like.

Execute Code: Use the course virtual environment: `source /course/cs1951a/venv/bin/activate`. To deactivate this virtual environment, simply type `deactivate`.

Option 3: Your Own Device (Not Recommended)

Set up virtual environment: You need to create a Python virtual environment and install `PySpark`. **Note:** `JDK8` is required (Spark Requirement) (https://spark.apache.org/docs/latest/).

Write Code: Use whatever editor or IDE you like.

Execute Code: Activate the virtual environment and run your program in the command line.

To test that you have set up everything correctly, we have provided a simple example where we used a `PySpark` MapReduce pipeline to do the task of counting the number of occurrences of each word in an input text. The code is in the `wordcount.py` file. Make sure that you can run this program, and feel free to play around / examine this file to understand how PySpark works.

Make sure to activate the virtual environment before running the `wordcount.py` file!

Submit Job to Dataproc Cluster:

After you pass the tests on the small and big datasets, you can try to run your recommendation system on a truly huge dataset. First upload your code to Google Cloud Shell. After that, the script `submit_job` is provided in the stencil to help you easily submit your job to the cluster. Since it is very expensive to run jobs on a huge GCP cluster, we won't keep the cluster running all the time during this assignment. Keep an eye out for a Piazza post detailing the schedule of availability for the course cluster.

Part 1: Introduction to MapReduce

30 points

In this part of the assignment you will solve two simple problems by making use of the `PySpark` library.

For each problem, you will turn in a python script (stencil provided) similar to `wordcount.py` that solves the problem using the supplied MapReduce framework, `PySpark` .

Problem 1: Inverted Index

Fill in the code for `inverted_index.py` , which creates an inverted index of a given set of documents. Given a set of documents, an inverted index is a matching from each word to a list of document ids of documents in which that word appears.

Your task is to design a MapReduce pipeline that would generate inverted indices for words in the given documents. You will have to think about how your data will move between the various stages of the pipeline and implement the following accordingly:

i. **`def mapper1(record)`**
ii. **`def reducer1(a, b)`**
iii. **`def mapper2(record)`**
iv. **`def reducer2(a, b)`**

As a note, you can feel free to use more mapper/reducer functions than those stated above, but you shouldn't need to - our solution manages to do it using only those 4 functions. In general, you have total control over the number of mapper and reducer functions that you use.

Your final task is to create such an inverted index matching with a MapReduce pipeline, using the mapper and reducer functions you just implemented. This query should return the inverted index of the given documents. You should use the variable **`inverted_index_result`** to store the result of your query.

For this problem, use the `books.json` and `books_small.json` datasets as the input to your pipeline. To run the file, execute the following command:

```
$ python inverted_index.py –d PATH/TO/data.json
```

where `PATH/TO/data.json` is the path to the json file with the data (so either ending in `books_small.json` or `books.json`). By default, without the `–d` flag, the data file path is `../data/mapreduce/books_small.json` .

Successfully running the script will create a file named `output_inverted_index.json` in a directory called `output` , which will contain the data that was collected by the pipeline in `inverted_index_result` . The format of the answer should look something like this:

```
[
    [
        "Answer",
        [
            "shakespeare-caesar.txt"
        ]
    ],
    ...
]
```

We provide you with a script to help you check the format of your json files:

```
$ ./check_format /PATH/TO/output_inverted_index.json
```

You can also verify the output of your pipeline on `books_small.json` using the provided `check_outputs_equal` script, passing it the path to your generated file and the ta solution's generated file, which can be found at `../data/mapreduce/ta_output/output_inverted_index.json` (or at `/course/cs1951a/pub/mapreduce/data/mapreduce/ta_output/output_inverted_index.json`):

```
./check_outputs_equal output/output_inverted_index.json ../data/mapreduce/ta_output/output_inverted_index.json
```

Problem 2: Relational Join

Fill in the code for `join.py`, where your task is to implement a SQL join query using a MapReduce pipeline. You will work with the data provided in `records.json` which contains tuples belonging to both 'Release' and 'Disposal' tables.

Consider the following SQL query:

```
SELECT *
FROM Release, Disposal
WHERE Release.CompanyID = Disposal.CompanyID
```

Your MapReduce query should produce the same output as the SQL query above. You can consider the two input tables, 'rele' and 'disp', as one big concatenated bag of records which gets fed into the map function record by record. **For each line/record in the json file, record[0] is the name of the table ({'rele', 'disp'}) and record[2] is CompanyID.** You will have to implement the following funtions:

i. **`def mapper1()`**
ii. **`def reducer()`**
iii. **`def mapper2()`**
iv. **`def filter (https://spark.apache.org/docs/latest/api/python/pyspark.html?highlight=flatmap#pyspark.RDD.filter)er()`**

You should use the variable **`join_result`** to store the result of your pipeline. Like above, you have total control over the number and order of functions in your pipeline; the above is just the order that our solution uses.

For this problem, use `records.json` as the input to your pipeline. Similar to Part 1, to run the file, activate the virtual environment and then execute the following command:

```
$ python join.py –d PATH/TO/records.json
```

This will create a file named `output_join.json` in your output directory, which will contain the data that was collected by the pipeline in `join_result`. It will look like:

```
[
    [
        "rele",
        "1995",
        "4836",
        ...,
        "disp",
        "2003",
        "4836",
        ...
    ],
    ...
]
```

You can also use this script to check the format.

```
$ ./check_format /PATH/TO/output_join.json
```

Part 2 - Movie Recommendation System

40 points

In Edwin Chen's blog article (http://blog.echen.me/2012/02/09/movie-recommendations-and-more-via-mapreduce-and-scalding/) on movie similarities, he describes how he used the Scalding (https://github.com/twitter/scalding) MapReduce framework to find similarities between movies. You will do the same by calculating the similarity of pairs of movies so that if someone watched Frozen (2013), you can recommend other movies they might like, such as Monsters University (2013).

Data

You are provided with a dataset of movie ratings:

> **Source:** MovieTweetings (https://github.com/sidooms/MovieTweetings) by Simon Dooms.
> **Overview:** Ratings are extracted from tweets and it contains up-to-date movie ratings (the earliest rating contained in this dataset is from Feb 28, 2013). It contains 571,073 ratings from 26,960 movies. And we will only be using a fraction of this.
> **Run Time:** Our implementation of `similarities.py` finishes executing in ~30 seconds on the small dataset and in ~45 seconds on the big one. It takes hours to finish executing on the huge dataset in Google Cloud Shell. But If you submit it to the cluster, it will complete the execution in ~5 minutes.

This dataset includes:

> **ratings.dat**
> - Format: `user_id::movie_id::rating`
> - Ratings are from 1 to 10 (whole-number ratings only)
> - 34,123 ratings (small)
> - 190,750 ratings (big)
> - 5, 714, 768 ratings (huge, already stored in Google Cloud)
>
> **movies.dat**
> - Format: `movie_id::movie_title`
> - Titles are identical to titles provided by the IMDB (including year of release)

Algorithm

As mentioned in Edwin Chen's blog article, we will use the different metrics between movie pairs to determine the similarity between them:

- For every pair of movies A and B, find all the people who rated both A and B.
- Use these ratings to form a Movie A vector and a Movie B vector.
- Calculate the similarity metrics between these two vectors.
- Whenever someone watches a movie, then you can recommend the most similar movies.

Like what Edwin did in his article, you will also experiment with four **similarity metrics**. The implementations for these metrics are provided in `similarities.py`. In the below equations, `n` is the number of users who rated both movie `X` and movie `Y`, `n1` is the number of users who rated movie `X`, and `n2` is the number of users who rated movie `Y`.

1. **Correlation (http://en.wikipedia.org/wiki/Correlation_and_dependence)**
   $Correlation(X, Y) = \frac{n\sum xy - \sum x \sum y}{\sqrt{n\sum x^2 - (\sum x)^2}\sqrt{n\sum y^2 - (\sum y)^2}}$
2. **Regularized Correlation**
   $Weight(X, Y) = \frac{n}{n + VirtualCount}$

   $RegularizedCorrelation(X, Y) = Weight(X, Y) * Correlation(X, Y) + (1 - Weight(X, Y)) * PriorCorrelation$

   As Edwin states, "we can also also add a regularized correlation, by (say) adding N virtual movie pairs that have zero correlation. This helps avoid noise if some movie pairs have very few raters in common (for example, The Great Gatsby had an unlikely raw correlation of 1 with many other books, due simply to the fact that those book pairs had very few ratings)."

   The stencil code uses VIRTUAL_COUNT = 10 and PRIOR_CORRELATION = 0.0, and you are welcome to experiment with different values **(but don't forget to change them back before you submit!)**
3. **Cosine Similarity (http://en.wikipedia.org/wiki/Cosine_similarity)**
   $Cosine(X, Y) = \frac{\sum xy}{\sqrt{\sum x^2}\sqrt{\sum y^2}}$
4. **Jaccard Similarity (http://en.wikipedia.org/wiki/Jaccard_index)**
   $Jaccard(X, Y) = \frac{n}{n_1 + n_2 - n}$

   As Edwin states, "recall that one of the lessons of the Netflix prize (http://blog.echen.me/2011/10/24/winning-the-netflix-prize-a-summary/) was that implicit data can be quite useful - the mere fact that you rate a James Bond movie, even if you rate it quite horribly, suggests that you'd probably be interested in similar action films. So we can also ignore the value itself of each rating and use a set-based similarity measure like Jaccard similarity."

Implementation

In `similarities.py`, you will implement a series of mappers and reducers. You will pass two input files, *ratings.dat* and *movies.dat*, to `similarities.py`, which will then output a list of **movie pairs** along with their similarity metrics between them like below (pretty-print JSON):

```
[
    [
        [
            "movie_title1",
            "movie_title2"
        ],
        [
            correlation_value,
            regularized_correlation_value,
            cosine_similarity_value,
            jaccard_similarity_value,
            n,
            n1,
            n2
        ]
    ],
    ...
]
```

For every pair of movies A and B, find all the people who rated both A and B and compute the number of raters for every movie. Then you can calculate 4 similarity metrics for every movie pair.

Below are the mappers and reducers that you will implement. We have provided the first part of the pipeline for you in the stencil code. For the remaining part, your MapReduce pipeline can have as many mappers and reducers as long as your outputs match the the two checkpoints and the final requirement.

Checkpoint 1

i. `def mapper0()`

ii. `def reducer()`

(Here you will be taking the parameters a and b and joining them. Don't overthink this! Refer to the lab if you get stuck.)

iii. `def mapper1()`

The output of your pipeline at this stage (after `mapper1`) should be of the following format:

```
[[key, value], [key, value], ...]
  where -
    key:   movie_title
    value: [ [user1_ID, user1_rating], [user2_ID, user2_rating], [user3_ID, user3_rating], ...]
```

The output at this stage should be stored in the variable `stage1_result` and will be written to the file `netflix_stage1_output.json`. This will serve as a checkpoint into your pipeline for the purposes of grading, so please make sure you implement this correctly. Its **format** will look like:

```
[
    [
        "$5 a Day (2008)",
        [
            [
                "22136",
                7
            ],
            ...
        ]
    ],
    ...
]
```

Note that the json file is very compact. If you want to pretty print it like above, you can use the following command. Don't worry about the order. It is because the collect() action is parallelized, and then the results are assembled. We will sort your results when grading.

```
$ cat PATH/TO/netflix_stage1_output.json  | python -m json.tool
```

You can use our script to check the format:

```
$ ./check_format /PATH/TO/netflix_stage1_output.json
```

You can also use our script to check the contents of your output on the small dataset:

```
$ ./check_outputs_equal PATH/TO/YOUR/netflix_stage1_output.json PATH/TO/TA/netflix_stage1_output.json
```

Checkpoint 2

Next, you are free to design your own MapReduce pipeline. Just don't forget to satisfy the requirement of the second checkpoint before the final output.

iv. `def mapper2()`

v. `def mapper3()`

vi. `...`

You are provided with implementations of 4 similarity metrics. You should refer back to the Algorithm section above to determine the input values for each of these metric functions. You will need to find the dot product between two vectors, the sum of each vector, the norm of each vector, and etc. In addition, you should ignore (do not include values for) movie pairs whose **regularized correlation** values are less than some threshold (i.e. 0.5) in order to keep only high value movie pairs.

The output of your pipeline at the second checkpoint should be of the following format:

```
[[key, value], [key, value], ...]
  where -
    key: movie_title1
    value: [[movie_title2, correlation_value, regularized_correlation_value, cosine_similarity_value, jaccar
d_similarity_value, n, n1, n2], [movie_title3, ..]]
```

IMPORTANT: You must (efficiently!) sort your data such that `movie_title2` s only occur for a `movie_title1` when `movie_title1 < movie_title2` (i.e. `movie_title1` comes alphabetically before `movie_title2`). The output at this stage should be stored in the variable `stage2_result` and will be written to the file `netflix_stage2_output.json` . This will serve as the second checkpoint into your pipeline for the purposes of grading, so please make sure you implement this correctly. Its **format** will look like:

```
[
    [
        "12 Years a Slave (2013)",
        [
            [
                "Jagten (2012)",
                0.6671378907298551,
                0.5221079144842344,
                0.9937391441268904,
                0.04265402843601896,
                36,
                617,
                263
            ],
            ...
        ]
    ],
    ...
]
```

You can use our script to check the format:

```
$ ./check_format /PATH/TO/netflix_stage2_output.json
```

You can also use our script to check the contents of your output on the small dataset:

```
$ ./check_outputs_equal PATH/TO/YOUR/netflix_stage2_output.json PATH/TO/TA/netflix_stage2_output.json
```

Final Output

vii. `def stageN()`

The output of the last stage should have the following format.

```
[[key, value], [key, value], ...]
  where -
    key: [movie_title1, movie_title2]
    value: [correlation_value, regularized_correlation_value, cosine_similarity_value, jaccard_similarity_va
lue, n, n1, n2]
```

The output of the last stage should be stored in the variable `final_result` , which will be written to the file `netflix_final_output.json` . Then the **format** of the output file will look like:

```
[
    [
        [
            "Captain America: The First Avenger (2011)",
            "Iron Man 3 (2013)"
        ],
        [
            0.7280290128482472,
            0.5824232102785978,
            0.9886495309825268,
            0.018682858477347034,
            40,
            82,
            2099
        ]
    ],
    [
        [
            "Captain America: The First Avenger (2011)",
            "The Avengers (2012)"
        ],
        [
            0.8188595535772019,
            0.603370197372675,
            0.990419871882812,
            0.0979020979020979,
            28,
            82,
            232
        ]
    ],
    ...
]
```

You can use our script to check the format:

```
$ ./check_format /PATH/TO/netflix_final_output.json
```

You can also use our script to check the contents of your output on the small dataset:

```
$ ./check_outputs_equal PATH/TO/YOUR/netflix_final_output.json PATH/TO/TA/netflix_final_output.json
```

We have provided a skeleton MapReduce query based on the TA solution; however, you are free to choose the internal implementation of your query. Please ensure that you adhere to the format of the data that you store in these three files: `netflix_stage1_output.json`, `netflix_stage2_output.json` and `netflix_final_output.json`.

How to Run (both locally and in Cloud Shell)

To test your program, first activate the virtual environment, and then enter:

```
$ python similarities.py –d PATH/TO/data
```

where `PATH/TO/data` is a path to the **folder** containing `movies.dat` and `ratings.dat`.

The default data path is `../data/mapreduce/recommendations/small/`. It will generate three json files in the folder `output` in your working directory.

Submit Your Job to the Dataproc Cluster

The clusters we've created for this assignment are very expensive to use, so please do your development using the small and big datasets and running locally or in the cloud shell. Only use the course cluster to run your code on the huge dataset once you've tested. The huge dataset has already been uploaded to the cluster, so you won't need to worry about the data files.

The script that submits your job to the course cluster is provided in the stencil code. First upload the script and your `similarities.py` to Google Cloud Shell. Then you can submit your PySpark job from Google Cloud Shell with the following command.

```
$ ./submit_job similarities.py
```

It will copy the generated JSON files from the course cluster to your Cloud Shell. The output folder is `output_huge` by default.

Written Questions

Please write your answers to the following questions in `written_questions.pdf` .

30 points (5 each)

1. Run `find_unique_movie_pair` on the final output of your pipeline on the huge dataset to find a unique movie pair to copy into your `written_questions.pdf` . You'll need to tell it where the final output json for the huge dataset is, as well as your cs login. For example, if your login was `abob00` , you might run:

   ```
   ./find_unique_movie_pair output/netflix_final_output.json abob00
   ```

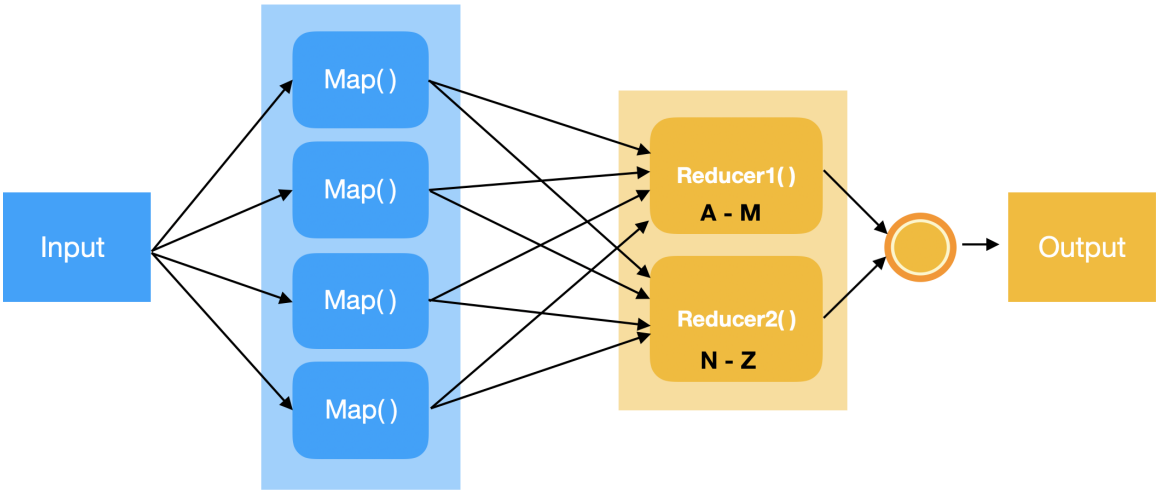
   And copy the output as your answer to this question.

2. Consider that you are given a dataset containing the details of babies born in the US in 2018. Each record is of the form `recordID :: year :: month :: state :: city` and there are around 3,978,497(4 million) records.

   In order to find the number of babies born during each month of the year, you come up with the following mapper and reducer (Refer wordcount.py) -

   ```
   Mapper: record -> (record.month, 1)
       For each record map the month to count 1.

   Reducer: k,[v] -> k, sum(list[v])
       For each key sum all values associated.
   ```



The MapReduce cluster provided to you consists of N mappers and but only 2 reducers as shown in the figure above.**Reducer1** receives all (key, value) pairs where keys are between A and M inclusive and **Reducer2** receives (key, value) pairs between N and Z inclusive.

Given that mapper and reducer function produces the correct output, **what possible issue(s)** could you face while processing a job consisting of 3,978,497 ~ 4 million records? Suggest a workaround for that issue.

3. You are given the following MapReduce pipeline which finds the 10 most frequent words beginning with each letter, in a large english text corpus.

```
Input: sentences

Mapper1 : sentence -> (word, count)
    for word in sentence:
        emit word, 1

Reducer1: list of (word, count) -> (letter, (word, count))
    total = 0
    for word, count in values:
        total += count
    emit word[0], (word, total)

Reducer2: list of (word, count) -> (letter, word)
    for word, count in values[:10]:
        emit letter, word
```

After testing on a small text file, it was noted that the pipeline does not produce correct output. Explain why this pipeline does not produce the correct output?