UE20CS322 Big Data Assignment 2

Implementation of Page Rank Algorithm with Page Embeddings

This is the second assignment for the UE20CS322 Big Data Course at PES University. The assignment consists of 2 tasks and focuses on running MapReduce jobs to implement a scenario of page rank that leverages graph embeddings.

Difficulty : Very Hard

The files required for the assignment can be found here.

Assignment Objectives and Outcomes

1. Learn how to run Iterative Map Reduce Jobs and use it for computing the famous Page Rank Algorithm.
2. At the end of this assignment, the student will be able to write and debug Page Rank code using Map Reduce.

Submission Deadlines

Phase I Submissions : `27/09/2022 11:59 PM IST`
Phase II Submissions : `01/10/2022 11:59 PM IST`

Exact portal timings will be announced later. Please do not wait until the last moment for submitting. Both Phase I and Phase II submissions will accept submissions for both the tasks. Phase II submissions will be the final round of submissions. There will be no extension of submissions dates and time. Please make sure to turn in your submissions before the due date.

Ethical practices

Please submit original code only. You can discuss your approach with your friends but you must write original code. All solutions must be submitted through the portal.
We will perform a plagiarism check on the code and you will be penalised if your code is found to be plagiarised.

Datasets

For this assignment we will be using two datasets. The first dataset consists of nodes that represent pages from berkely.edu and stanford.edu domains and directed edges represent hyperlinks between them. The second dataset is also a graph

dataset that contains network of hyperlinks from a snapshot of Google Web Graph from 2002.

The datasets can be downloaded from Berkley-Stanford and Google Web Graph.

Each line of the dataset consists of two values, the `source page` and the `destination page` separated by `\t`. The pages are denoted using a numerical ID. An edge from `x` to `y` indicates a hyperlink on page `x` to page `y`. The dataset may look like the following :

```
12 6
21 32
60 72
15 3
13 6
49 6
10 11
```

Software/Languages to be used:

1. Python `3.10.x`
2. Hadoop `v3.3.3` only

Submission Link

Portal for Big Data **RR Campus** Assignment Submissions.

Portal for Big Data **EC Campus** Assignment Submissions.

Submission Guidelines

You will need to make the following changes to your mapper and reducer scripts to run them on the portal

1. Include the following shebang on the first line of your code

```
#!/usr/bin/env python3
```

2. Convert your files to an executable

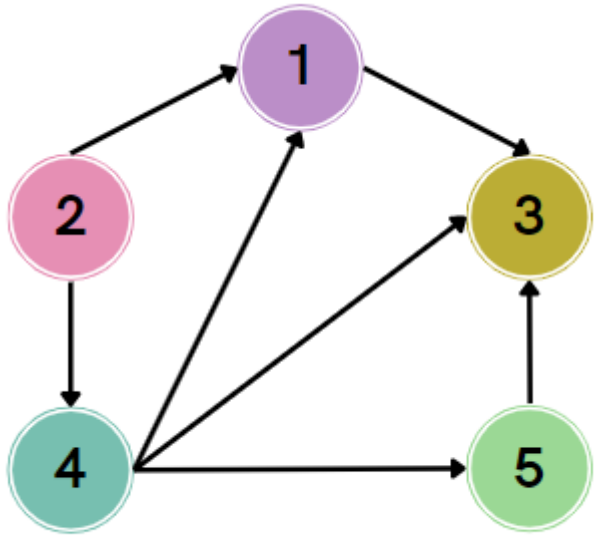
```
chmod +x mapper.py reducer.py
```

3. Convert line breaks in DOS format to Unix format (**this is necessary if you are coding on Windows** - your code will not run on our portal otherwise)

```
dos2unix mapper.py reducer.py
```

Task Specifications

The following graph will be used as an example to explain the sample input and outputs.



Task 1

Problem Statement

Converting the nodes in the input dataset to Adjacency List Representation using Map Reduce.

Dataset to be used for this task is Berkley-Stanford.

Description

Write Mapper and Reducer scripts that reads the input dataset through `stdin` processes it and generates the adjacency list representation of the input graph. Alongside, you must also generate the initial page ranks for all the nodes that have out going edges in the graph.

The mapper is responsible for reading input dataset through `stdin`, processing the input and generating intermediate key value pairs. The reducer then takes in these key value pairs through `stdin` and writes the adjacency list to HDFS.

The reducer shall also write the initial page rank to a file stored locally.

Input Format

The Mapper file takes in the input dataset through `stdin`. The input data may not be sorted, but it will be grouped by nodes. The Reducer takes in the intermediate key value pairs as input through `stdin` and a **command line argument** that specifies the **absolute path to `w` file**. The `w` file is stored locally and contains the initial page ranks for all the source nodes in the adjacency list. As a preprocessing step, we would like you to make mapper **ignore all those lines that start with a `#`.**

Output Format

Display each node in the network along with its adjacent nodes. The output from the reducer *may* look like the following. The separator between the `from_node_id` and `list_of_adj_nodes` has to be `'\t'`. The output should be sorted in **lexicographical** order of `from_node_id`.

```
from_node_id    list_of_adj_nodes
```

The initial page ranks should be written locally to a new file called `w` (**to be strictly followed**). The values are comma separated and newline delimited. The output should be sorted in **lexicographical** order of `node`.

```
node,pagerank
```

Implementation Guidelines

1. The adjacency list should be written to **HDFS**, and the page rank vector should be written locally in a file called `w`.
2. The path to the `w` file will be passed as a command line argument to the reducer file.
3. Never load the whole dataset into memory. It is guaranteed that loading the whole dataset to memory will exceed memory limits.
4. It is possible to generate the adjacency list without explicitly creating the adjacency list in memory. Your solutions must have **O(1) Space Complexity**. If your solution has higher Space Complexity, then your solution will mostly likely exceed either time limits or memory limits.
5. You are not allowed to use any sorting functions in your sripts.

6. Time Limit for this task would be 30s. Please ensure your code runs on the **complete dataset** under 30s. If your code takes more than 30s, you will get a TLE and 0 marks will be given.
7. Exceeding memory limits, will cause our containers to crash. Repeating such errors would result in the team being blacklisted for few hours.

Helpful Commands

You are required to use Hadoop to run your codes. Using the python commands for this assignment will result in wrong answer.

Kindly add your datasets to HDFS using the following command :

```
hdfs dfs -put /path_to_dataset_on_local_disk /path_in_HDFS
```

Make your scripts executables by using :

```
sudo chmod +x mapper.py reducer.py
```

Commands to execute the mapper and reducer in hadoop would be as shown below :

```
hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-3.3.3.jar \
-mapper "/absolute_path_to_mapper.py" \
-reducer "'/absolute_path_to_reducer.py' '/absolute_path_to_w'" \
-input "/path_to_dataset_on_HDFS/dataset.txt" \
-output "/path_to_output_on_HDFS"
```

Note : Replace `dataset.txt` with the actual filename of the dataset. Also replace `path_to_streaming.jar` if you have the jar file stored somewhere else.

Example

1. Input network

```
1 3
2 1
2 4
4 5
4 3
5 3
4 1
```

2. `w` file containing initial page ranks, written locally

```
1,1
2,1
4,1
5,1
```

3. Output file containing adjacency list, written to HDFS

```
1   [3]
2   [1, 4]
4   [5, 1, 3]
5   [3]
```

Note that the nodes in the adjacency list need to be in the order that Hadoop returns. The nodes must not be sorted manually.

Task 2

Problem Statement

Implementing the famous Page Rank Algorithm using Iterative Map Reduce Jobs.

The dataset to be used for this task is Google Web Graph

Description

In this task you will be using the code developed in the first task to generate the adjacency list for Google Web dataset. The dataset will be stored in HDFS in the form of adjacency list representation of the graph. You are required to rerun your Task 1 code to generate the adjacency list for the Task 2 dataset.

Once the adjacency has been created and stored in HDFS, Task 2 requires you to use that adjacency list as input to Mapper file along with Page Embeddings for the same and the `w` file and generate intermediate key value pairs for the reducer. The reducer then takes in these key value pairs and computes the ranks and writes the page ranks to a new `w` file. This new `w` file must also contain pagerank for the nodes with no outgoing links as well. The `page_embeddings` stores the embeddings for each page in the graph. This embeddings is a vector of size `6`.

The input to the mapper file would be the adjacency list taken from `stdin` and command line arguments - **path to `w` file and `page_embedding` file in the same order.**

The mapper will read the `adjacency list`, `w` file and the `page_embeddings` and the reducer will compute the new page ranks based on the given equations.

(1) $Rank(p) \ = \ 0.34 \ + \ 0.57 \sum Contribution\ of\ nodes\ pointing\ to\ p$

where,

(2) $Contribution(p,\ q) \ = \ \frac{Rank'(p).Similarity(p,q)}{Number of outgoing\ links from\ p}$

where $Rank'(p)$ is the *previous* rank of `p`, `p` is a node pointing to `q`, and

(3) $Similarity(p,\ q) \ = \ \frac{\overrightarrow{p}.\overrightarrow{q}}{|\overrightarrow{p}|^2 + |\overrightarrow{q}|^2 - \overrightarrow{p}.\overrightarrow{q}}$

where $\overrightarrow{p}$ and $\overrightarrow{q}$ are the vectors for page `p` and page `q` respectively which can be obtained from the `page_embeddings` file.

Implementation Guidelines

1. We will provide a bash script that will perform the following operations:
    - Mapper reads the `adjacency list`, `w` and `page_embeddings` file and computes contributions
    - The `adjacency list` is read from HDFS
    - The `page_embeddings` and `w` file are read locally, the paths to which are provided as command line arguments
    - Each page's embedding is a vector of size `6`, and this size will be fixed for all testcases
    - Reducer computes new page ranks and writes output to `w1`
    - If values of `w` and `w1` are nearly similar (i.e, has reached convergence), exit
    - Else:
        - Delete `w` and rename `w1` to `w`
        - Redo from step 1
2. Reaching convergence means that the difference between the updated page ranks and the previous for every page should be < `CONVERGENCE_LIMIT`
3. The value of `CONVERGENCE_LIMIT` will be decided by the bash script.
4. All ranks are to be **rounded off to 2 decimal places**. The **rounding off should only be done while printing to `STDOUT`** and not during computation.

Similarity Function Implementation

In this section we will give a brief idea about how we would like you to implement the similarity function. This is done to ensure that we have a consistent implementation of similarity function. The idea discussed below takes few ideas from loop optimizations when calculating dot products which ensures that your implementation is as fast as possible.

Loop Optimization

Loop Optimizations are widely used when we want to reduce the number of times a loop iterates. The way python loops works makes using loops inefficient. Python packages the loop condition and the loop body and passes it to C for execution during runtime. This adds a lot of unnecessary overhead to your code. One way to mitigate this is to make the loop body execute more iterations at once. This technique is called as loop unrolling. Below is a small example on how to implement loop unrolling.

Let's say we are summing up 1000 numbers thats stored in a list. The code for doing that without using loop unrolling would be as follows.

```
n = 1000
numbers = list(range(0,n))
sum = 0
i = 0
while i < n:
    sum += numbers[i]
    i += 1

print(sum)
```

This method works fine when `n` is smaller. But when `n` becomes larger, python loop overhead becomes a major bottleneck. To solve this we use loop unrolling. The following code shows to sum up the same 1000 numbers using loop unrolling technique.

```
n = 1000
numbers = list(range(0,n))
sum = 0
i = 0
kernel_size = 4
bound = n - kernel_size + 1

```

```
while i < bound:
    sum += numbers[i]
    sum += numbers[i+1]
    sum += numbers[i+2]
    sum += numbers[i+3]
    i += kernel_size

while i < n:
    sum += numbers[i]
    i += 1

print(sum)
```

In the above code, we have used loop unrolling to reduce the loop packaging overhead by increasing the loop body. Instead of performing a single iteration in loop body, we perform 4 iterations in a single body. More specifically, we perform `kernel_size` worth of iterations. This `kernel_size` becomes a tuning parameter that needs to be optimized for the problem statement at hand. The second loop is used to handle the edge cases where the numbers are not divisible by `kernel_size`. We calculate the bound upto which we can move at strides of `kernel_size` and loop only upto that bounds. The remaining elements must be summed up normally. Thus we need to tune `kernel_size` so that we minimise the number of iterations of the second loop.

Similarity Function

We are using Jaccard Similarity function which is described previously. Here the input parameters to the similarity function are two vectors `p` and `q`. Both these vectors are of size `6`. In real world, these vectors could be of much higher dimensions. Hence we would like you to use loop unrolling that you learnt above while computing the dot products of `p` and `q`. Vector `p` is the `page_embeddings` of the source vertex in the adjacency list. Vector `q` is the `page_embeddings` of each node in the `adjacency_list[source_vertex]`.

You need to apply loop unrolling techinque described above while computing the dot products of `p` and `q`. You also need to apply this technique while calculating the Norm of the vectors `p` and `q`. Another optimization we would like you to perform is as follows. Notice that we are calculating the Norm of `p` for every node in the `adjacency_list[p]`. If the length of list is huge, then we will spend a lot of time computing the same Norm of `p`. It would be ideal if we could cache the value of Norm of `p` and use the same value in subsequent iterations. Summing everything up, we would like to use the following structure when implementing your similarity function.

```
def similarity(p, q, cache):
# Perform some initializations as required
```

```
# Calculate the correct bounds using appropriate kernel_size
# (think about how big the size of p is to determine the ideal value).

if cache is None:
        # Compute Dot product, Norms of p and q using loop unrolling.
        # (Note you can compute everything in one loop unrolling segment).
        cache = Norm of p
else:
        # Compute Dot product, Norm of q using loop unrolling.
        # (Note you can compute everything in one loop unrolling segment).

# Using the cache and Norm of q and the dot products, calculate similarity
return similarity, cache  # pass the same cache again in subsequent calls.
```

When the source vertex changes, we need to clear our cache and recompute Norm of `p` as `p` is a new vertex.

Note : If you don't follow these steps, your submission will likely be exceeding time limits.

Input Format

The mapper will receive two command line arguments: the absolute path to the `w` file and the absolute path to the `page_embeddings` file. The adjacency list must be taken through `stdin`.

Output Format

For each page in the network, display the page's ID along with its updated page rank on a single line. The values are comma separated and newline delimited. The output should be sorted in **lexicographical** order of `node`.

```
node,pagerank
```

Helpful Commands

You are required to use Hadoop to run your codes. Using the python commands for this assignment will result in wrong answer.

Kindly add your datasets to HDFS using the following command :

```
hdfs dfs -put /path_to_dataset_on_local_disk /path_in_HDFS
```

Make your scripts executables by using :

```
sudo chmod +x mapper.py reducer.py
```

Brief Explanation on how to run iterative MR Job. This section will explain the different parts of `iterate-hadoop.sh` file.

```
#!/bin/sh
CONVERGE=1
ITER=1
rm w w1 log*
$HADOOP_HOME/bin/hadoop dfsadmin -safemode leave
hdfs dfs -rm -r /task-*
```

Performs some basic initializations by setting `CONVERGE=1` and `ITER=1`. `CONVERGE` tells whether we have converged or not and `ITER` specifies the current iteration of MR job. Then on the last few lines we remove the previous `w, w1 & log` files.

Now we execute Task 1. Remeber that you need to use the Google dataset for this. The following snippet is for demonstration purposes only. Don't forget to update the paths in the following command accordingly.

```
$HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streamin
g-3.3.3.jar \
-mapper "'LOCAL_PATH_to_Task1_Mapper_file'" \
-reducer "'LOCAL_PATH_to_Task_1_Reducer_file' 'LOCAL_PATH to_w_file'"  \
-input /HDFS_PATH_to_input.txt \
-output /task-1-output
```

Once we have the adjacency list from Task 1, we are ready to start with Task 2.

```
while [ "$CONVERGE" -ne 0 ]
do
        echo "############################## ITERATION $ITER ##################
##########"
        $HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-
```

```
streaming-3.3.3.jar \
        -mapper "'LOCAL_PATH_to_Task_2_Mapper_file' 'LOCAL_PATH_to_w_file' 'LOC
AL_PATH_to_page_embeddings_JSON_file'" \
        -reducer "'LOCAL_PATH_to_Task_2_Reducer_file'" \
        -input HDFS_PATH_to_input \
        -output /task-2-output
        touch w1
        hadoop dfs -cat /task-2-output/part-00000 > "LOCAL_PATH_to_current_dire
ctory/w1"
        CONVERGE=$(python3 LOCAL_PATH_to_current_directory/check_conv.py $ITER>
&1)
        ITER=$((ITER+1))
        hdfs dfs -rm -r /task-2-output/
        echo $CONVERGE
done
```

The explanation for the above snippet is as follows. We loop until we have not converged. In each iteration, we schedule a new MR Job by passing the appropriate mapper and reducer files. The output of the MR Job would be stored in HDFS. This output would be the new `w` file with updated page ranks. We then copy this file to a new local file `w1`. Both the files are then passed to a python script that checks if the ranks for pages have converged. If ranks have converged we stop the iteration and if not converged we go to the next iteration. Finally, before we start our next iteration, we need to remove the output folder in HDFS.

Note : The above snippets are only for demonstration. You are required to change `iterate-hadoop.sh` script. You need to update all the paths to mapper and reducer files accordingly. You are also required to update the paths to `-input` and `-output` and other commands that use the output paths. By now, we hope that you have a fairly good understanding of how HDFS and Hadoop works and you are expected to change the paths in script file by yourselves.

Once you have identified and modified the changes that need to be done in the `iterate-hadoop.sh` file, you can exceute Task 2 by the following command :

```
bash iterate-hadoop.sh
```

Example

Consider the following to be the input `page_embeddings` for the provided sample network with 5 pages.


```
{
    "1": [
        -0.5937666,
        0.684082,
        -0.5772033,
        0.3481369,
        0.0965215,
        0.3667577
    ],
    "2": [
        0.7946288,
        -0.4162117,
        0.1517516,
        -0.4744227,
        -0.193617,
        0.3375438
    ],
    "3": [
        -0.8574042,
        0.2909393,
        0.745526,
        0.5061621,
        -0.1202947,
        0.392672
    ],
    "4": [
        -0.3609459,
        -0.0422608,
        -0.9533574,
        -0.4942852,
        0.1140913,
        0.4222589
    ]
```

```
    "5": [
        0.7639189,
        0.4191339,
        -0.1799131,
        -0.0183615,
        0.4972066,
        0.961261
    ]
}
```

Attached below are the **initial** `page ranks` for the provided network.

```
1,1
2,1
4,1
5,1
```

Here is the `adjacency list`

```
1   [3]
2   [1, 4]
4   [5, 1, 3]
5   [3]
```

The above `adjacency list` can be converted to the following matrix `M` where `M[i][j]` stores the initial contribution of page `i` to page `j` **before the similarity scores have been multiplied**.

#	page 1	page 2	page 3	page 4	page 5
page 1	0	0	1	0	0
page 2	0.5	0	0	0.5	0
page 3	0	0	0	0	0
page 4	0.33	0	0.33	0	0.33
page 5	0	0	1	0	0

As mentioned in equation `3`, the expected similarity matrix `S` will look like this, where `S[i][j]` is the similarity between pages `i` and `j` using the vectors obtained from the `page_embeddings` file.

#	page 1	page 2	page 3	page 4	page 5
page 1	1.0	-0.26	0.22	0.34	0.11
page 2	-0.26	1.0	-0.2	-0.02	0.25
page 3	0.22	-0.2	1.0	-0.14	-0.09
page 4	0.34	-0.02	-0.14	1.0	0.11
page 5	0.11	0.25	-0.09	0.11	1.0

Further, we can obtain the final contribution matrix `C` where `C[i][j]` contains the contribution `M[i][j]` multiplied by `S[i][j]`.

#	page 1	page 2	page 3	page 4	page 5
page 1	0	0	0.22	0	0
page 2	-0.13	0	0	-0.01	0
page 3	0	0	0	0	0
page 4	0.112	0	-0.046	0	0.036
page 5	0	0	-0.09	0	0.0

Replacing the values of `p` and `q` as as `page 2` and `page 1` respectively in the equation `2`, we obtain the initial contribution of `page 2` to `page 1` as the following:

Initial page rank of `page 2` : 1
Number of outgoing links from `page 2` : 2
Initial contribution = 1/2 = 0.5

Multiplying the initial contribution of `page 2` to `page 1` with the similarity score between the two pages obtained from matrix `S`, we get the complete contribution of `page 2` to `page 1` as the following:

Initial contribution = 1/2 = 0.5
Similarity between `page 2` and `page 1` = -0.26
Complete contribution = -0.13

The above process can be repeated to generate the values in all the cells of the matrices `M`, `S` and `N`.

Hence, the final page rank of `page 2` is given by equation `1` where,

New page rank of `page 2` after **one iteration** = 0.34 + 0.57 x (contribution of `None`) = 0.34 + 0.57 x (0) = 0.34

Here, it is 0.57 x (contribution of `None`) as `page 2` has no incoming edges

The updated page ranks are calculated for all pages to obtain the following result in the `w1` file.

```
1,0.33
2,0.34
3,0.39
4,0.33
5,0.36
```

Note that node `3` is also present in the new pagerank `w1` file. You are required to calculate the page rank for nodes with no outgoing links. The page rank for all nodes will converge to a value after performing the above steps for some iterations.

Good Luck!