CSE6242 / CX4242: Data and Visual Analytics | Georgia Tech | Spring 2016

Homework 3 : Hadoop, Spark, Pig and Pandas

Due: Friday, April 1, 2016, 11:55 PM EST

Prepared by Gopi Krishnan Nambiar, Nilaksh Das, Pradeep Vairamani, Ajitesh Jain, Vishakha Singh, Polo Chau

Submission Instructions:

It is important that you read the following instructions carefully and also those about the deliverables at the end of each question or you may lose points.

- ❏ Submit a single zipped file, called "HW3-{YOUR_LAST_NAME}-{YOUR_FIRST_NAME}.zip" containing all the deliverables including source code/scripts, data files, and readme. Example: 'HW3-Doe-John.zip' if your name is John Doe. Only .zip is allowed (no .rar, etc.)
- ❏ You may collaborate with other students on this assignment, but you must write your own code and give the explanations in your own words, and also mention the collaborators' names on T-Square's submission page. All GT students must observe the honor code. **Suspected plagiarism and academic misconduct will be reported and directly handled** by the Office of Student Integrity (OSI). Here are some examples similar to Prof. Jacob Eisenstein's NLP course page (grading policy):
  - ❏ **OK:** discuss concepts (e.g., how cross-validation works) and strategies (e.g., use hashmap instead of array)
  - ❏ **Not OK:** several students work on one master copy together (e.g., by dividing it up), sharing solutions, or using solution from previous years or from the web.
- ❏ If you use any "*slip days*", you must write down the number of days used in the T-square submission page. For example, "Slip days used: 1". Each slip day equals 24 hours. E.g., if a submission is late for 30 hours, that counts as 2 slip days.
- ❏ At the end of this assignment, we have specified a folder structure of how to organize your files in a single zipped file. 5 points will be deducted for not following this strictly.
- ❏ Wherever you are asked to write down an explanation for the task you perform, stay within the word limit or you may lose points.
- ❏ In your final zip file, do not include any intermediate files you may have generated to work on the task, unless your script is absolutely dependent on it to get the final result (which it ideally should not be).
- ❏ After all slip days are used up, 5% deduction for every 24 hours of delay. (e.g., 5 points for a 100-point homework)
- ❏ We will not consider late submission of any missing parts of an homework assignment or project deliverable. To make sure you have submitted everything, download your submitted files to double check.

Applying for AWS Educate Account (do this now!) & SET UP ALERTS

It is **EXTREMELY IMPORTANT** that you apply for an “AWS Educate” account RIGHT AWAY to get $100 free credits, and verify that the credit has been properly applied on your account, so that you can work on Task 3. Creating the account can take days and HW3’s computation can take hours to run, so if you do not do this now, you may jeopardize your HW3 progress.

- Go to https://aws.amazon.com/education/awseducate/
- Click the **Apply Now** button on the right
- Click the **Apply for AWS Educate for Students** button
- Choose **student**, and click **Next**
- Fill out the application
    - If you do not have an AWS account ID, you will need to sign up to get one, as the form suggests.
    - To find your AWS account ID, log into the console and then going to this link https://console.aws.amazon.com/billing/home?#/account
    - Your AWS Account ID is right at the top of this screen

Also it is EXTREMELY IMPORTANT that you set up a billing alarm on AWS to notify you when your credits are running low. Here’s how:
http://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/free-tier-alarms.html

Shut down **EVERYTHING** when you're done with the instances (don’t leave them on over spring break!), or you may get surprising credit card bills. Highest record from previous classes went above $2000! The good news is you can call AWS to explain the situation and they should be able to waive the charges.

Setting up Development Environment for Task 1 and Task 2

Installing CDH

Download a preconfigured virtual machine (VM) image from Cloudera (CDH).
Please use 5.3.x version of CDH or higher.
You can choose any VM platform, but we recommend VirtualBox; it is free.
Instructions to setup and configure the VM can be found here.

Read these important tips to improve the VM's speed and reduce memory consumption.

Once you launch the VM, you will have a GUI environment with cloudera user, which has administrator (sudo) privilege. The account details are:

username: cloudera
password: cloudera

The virtual image comes pre-installed Hadoop and Spark. You will use them for this HW.

Tip: You may want to setup port forwarding to obtain SSH access of guest operating system.

Loading Data into HDFS

Now, let's load our dataset into the HDFS (Hadoop Distributed File System), which is an abstract file system that stores files on clusters. Your Hadoop or Spark code will directly access files on HDFS. Paths on the HDFS look similar to those on the UNIX system, but you can't explore them directly using standard UNIX commands. Instead, you need to use ***hadoop fs*** commands. For example

```
hadoop fs -ls /
```

Download the following two graph files: graph1.tsv[1] (~5MB) and graph2.tsv[2] (~873MB). Use the following commands to setup a directory on the HDFS to store the two graph datasets. Please do not change the directory structure below (/user/cse6242/) since we will grade your homework using the scripts which assume the following directory structure.

```
sudo su hdfs
hadoop fs -mkdir /user/cse6242/
hadoop fs -chown cloudera /user/cse6242/
exit
su cloudera
```

[1] This graph is originally from the Enron email network data set. There are 321 thousand edges and 77 thousand nodes.
[2] This graph is from the Portuguese Wikipedia link data set. There are 53 million edges and 1 million nodes.

```
hadoop fs -put graph1.tsv /user/cse6242/graph1.tsv
hadoop fs -put graph2.tsv /user/cse6242/graph2.tsv
```

Now both files - graph1.tsv and graph2.tsv are on the HDFS at **/user/cse6242/graph1.tsv** and **/user/cse6242/graph2.tsv**

Setting up Development Environments

We found that compiling and running Hadoop/Scala code can be quite complicated. So, we have prepared some skeleton code, compilation scripts, and execution scripts for you that you can download here. You should use this package to submit your homework.
This packaged zip file has preset directory structures. As you will zip all the necessary files with the same directory structure in the end, you may not want to modify the structure. (See the end of this document for details.) In the directories of both *Task1* and *Task2*, you will find **pom.xml**, **run1.sh, run2.sh** and the **src** directory.

- the **src** directory contains a main Java/Scala file that you will primarily work on. We have provided some code to help you get started. Feel free to edit it and add your files in the directory, but the main class should be Task1 and Task2 accordingly. Your code will be evaluated using the provided **run1.sh** and **run2.sh** file (details below).

- **pom.xml** contains the necessary dependencies and compile configurations for each task. To compile, you can simply call Maven in the corresponding directory (Task1 or Task2 where pom.xml exists) by this command:

  ```
  mvn package
  ```

  It will generate a single JAR file in the target directory (i.e. target/task2-1.0.jar). Again, we have provided you some necessary configurations to simplify your work for this homework, but you can edit them as long as our run script works and the code can be compiled using mvn package command.

- ***run1.sh***, ***run2.sh*** are the script files that run your code over graph1.tsv (run1.sh) or graph2.tsv (run2.sh) and download the output to a local directory. The output files are named based on its task number and graph number (e.g. task1output1.tsv). You can use these run scripts to test your code. Note that these scripts will be used in grading.

Here's what the above scripts do (you can open them in a text editor to see what is going on):

1. Run your JAR on Hadoop/Scala specifying the input file on HDFS (the first argument) and output directory on HDFS (the second argument)
2. Merge outputs from output directory and download to local file system.
3. Remove the output directory on HDFS.

[25pts] Task 1: Analyzing a Large Graph with Hadoop/Java

a) [15 pts] Writing your first simple Hadoop program

Please first go over the Hadoop word count tutorial to get familiar with Hadoop and some Java basics. You will be able to complete this task with only some knowledge about Java[3].

Goal

Your task is to write a MapReduce program in Java to calculate the maximum of the weights of all **outgoing** edges for each node in the graph.

You should have already loaded two graph files into HDFS. Each file stores a list of edges as tab-separated-values. Each line represents a single edge consisting of three columns: (source node ID, target node ID, edge weight), each of which is separated by a tab (\t). Node IDs are positive integers, and weights are also positive integers. Edges are ordered randomly.

```
src   tgt   weight
51    117   1
51    194   1
51    299   3
151   230   51
151   194   79
130   51    10
```

Your code should accept two arguments upon running. The first argument (args[0]) will be a path for the input graph file on HDFS, and the second argument (args[1]) will be a path for output directory. The default output mechanism of Hadoop will create multiple files on the output directory such as part-00000, part-00001, which will be merged and downloaded to a local directory by the supplied run script. Please use the run scripts for your convenience.

The format of the output should be as follows. Each line represents a node ID and the maximum of the weights of its outgoing edges' weights. The ID and the maximum weight must be separated by a tab (\t). Lines do not need be sorted. The following example result is computed based on the toy graph above. Please exclude nodes that do not have outgoing edges.

```
51    3
151   79
130   10
```

[3] Some of you may ask "Why should I learn Java?" A main reason is that most (fast) production code is written in C++ or Java. For enterprise software, Java is extremely common used.

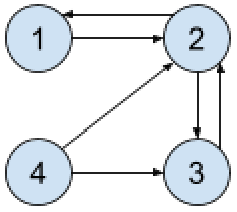
Test your program on graph1.tsv and graph2.tsv. Explain your MapReduce procedure(s). *Using the above example*, trace the input and output of your map and reduce functions, i.e., given the above graph as the input, describe and explain the input and output of your map and reduce function(s). Write down your answers in **description.pdf**. You are welcome to explain your answers using a combination of text and images.

b) [10 pts] Designing a MapReduce algorithm (and thinking in MapReduce)

Design a MapReduce algorithm that accomplishes the following task: for each node i in a **directed** graph G, find that node's **out neighbors' out neighbors**. Node *v* is an out neighbor of node *u* if there is a directed edge pointing from node *u* to node *v*. In other words, your task is find every "2-hop-away" neighbor of every node i in the graph G; such a neighbor is connected by at least one directed path of length 2 that originates from node i.

NOTE: You only need to submit pseudo code, a brief explanation of your algorithm and trace of input and output of your map and reduce functions for the graph given below. No coding is required.

Consider the following toy graph:



Input of your algorithm:

```
src  tgt
4    3
1    2
2    3
4    2
2    1
3    2
```

Output of your algorithm should be:

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

Here, the output pair (*u, v*) indicates *v* is an out neighbor of an out neighbor of *u*. Also *u* and *v* are distinct nodes, i.e., your output **should not** contain pairs such as (1,1), (2,2) and (3,3). However, your output **can** contain duplicate pairs. For example, pair (1, 3) may occur more than once in your algorithm's output.

Here we explain the example results further:

Output		*Path*	*Detailed description*
4	1	4->2->1	1 is an out neighbor of 2; 2 is an out neighbor of 4
1	3	1->2->3	3 is an out neighbor of 2; 2 is an out neighbor of 1
4	2	4->3->2	...
3	1	3->2->1	...
4	3	4->2->3	...

Explain your algorithm; using **the above toy graph,** trace the input and output of your map and reduce functions, i.e., given the above graph as the input, describe and explain the input and output of your map and reduce function(s). Write down your answers in **description.pdf**. You are welcome to explain your answers using a combination of text and images.

Deliverables

1. **[5 pts] Your Maven project directory including Task1.java.** Please see detailed submission guide at the end of this document. **You should implement your own MapReduce procedure and should not import external graph processing library.**

2. **[2 pts] task1output1.tsv**: the output file of processing graph1 by run1.sh.

3. **[3 pts] task1output2.tsv**: the output file of processing graph2 by run2.sh.

4. **[15 pts] description.pdf**: Answer for parts a and b. Your answers should be written in 12pt font with at least 1" margin on all sides. Your pdf must not exceed 2 pages.

[25pts] Task 2: Analyzing a Large Graph with Spark/Scala

Please go over this Spark word count tutorial to get more background about Spark/Scala.

Goal

Your task is to cascade the edge weights in graph1.tsv and graph2.tsv to node weights, and finally determine the accumulated node weights using Spark, in Scala. Assume that 80% of the edge weight comes from the source node and 20% from the target node. When loading the edges, parse the edge weights using the *toInt* method and before cascading, filter out (ignore) all edges whose edge weights equal 1. That is, only consider edges whose edge weights do not equal 1.

Consider the following example:

Input:
```
src   tgt   weight
1     2     40
2     3     100
1     3     60
3     4     1
```

Output:
```
1     80.0         = 0.8*40 + 0.8*60
2     88.0         = 0.2*40 + 0.8*100
3     32.0         = 0.2*100 + 0.2*60
```

Notice here that the edge from `3` to `4` is ignored since its weight is 1.

Your Scala program should handle the same two arguments as in Task 1 for input and output from the console, and should generate the same formatted output file on the supplied output directory (tab-separated-file). Please note that the default Spark *saveastextfile* method uses saving format that is a different from Hadoop's, so you need to format the result before saving to file (Tip: use *map* and *mkString*). The result doesn't need to be sorted.

Based on your approach, you may find some of the following functions helpful:
map, *reduce*, *reduceByKey*, *union*, *cogroup, filter*, *join*, *flatMap*, *groupByKey*, *intersection*

You can refer to the full list of RDD what is RDD? functions here.

Deliverables

1. **[10 pts] Your Maven project directory including Task2.scala**. Please see the detailed submission guide at the end of this document. **You may not use any external graph processing libraries.**

2. **[2 pts] task2output1.tsv**: the output file of processing graph1 by run1.sh.

3. **[3 pts] task2output2.tsv**: the output file of processing graph2 by run2.sh.

4. **[10 pts] description.txt**: describe your approach and refer to line numbers in your code to explain how you're performing each step in not more than 150 words.

[35pts] Task 3: Analyzing Large Amount of Data with Pig on AWS

You will try out PIG (http://pig.apache.org) for processing n-gram data on Amazon Web Services (AWS). This is a fairly simple task, and in practice you may be able to tackle this using commodity computers (e.g., consumer-grade laptops or desktops). However, we would like you to use this exercise to learn and solve it using distributed computing on Amazon EC2, and gain experience (very helpful for your future career in research or industry), so you are prepared to tackle more complex problems.

The services you will primarily be using are Amazon S3 storage, Amazon Elastic Cloud Computing (EC2) virtual servers in the cloud, and Amazon Elastic MapReduce (EMR) managed Hadoop framework.

This task will ideally use up only a very small fraction of your $100 credit. AWS allows you to use up to 20 instances in total (that means 1 master instance and up to 19 core instances) without filling out a "limit request form". **For this assignment, you should not exceed this quota of 20 instances.** You can learn about these instance types, their specs, and pricing at
https://aws.amazon.com/ec2/instance-types/
https://aws.amazon.com/ec2/pricing/
(In the future, for larger jobs, you may want to use AWS's pricing calculator:
http://calculator.s3.amazonaws.com/index.html)

Please read the AWS Setup Guidelines provided to set up your AWS account. In this task, you will use subsets of the Google books *n-grams* dataset (full dataset is here), on which you will perform some analysis. An '*n*-gram' is a phrase with *n* words; the full n-gram dataset lists n-grams present in the books on books.google.com along with some statistics.

You will perform your analysis on two custom datasets, extracted from the Google books bigrams (2-grams), that we have prepared for you: a large one (**s3://cse6242-bigram-big**) and a smaller one (**s3://cse6242-bigrams-small**).

NOTE: Both these datasets are in the **US-Standard (US-East)** region. Using machines in other regions for computation would incur data transfer charges.

The files in these two S3 buckets are stored in a tab (\t) separated format. Each line in a file has the following format:

```
n-gram TAB year TAB occurrences TAB books NEWLINE
```

An example for 2-grams (or bigram) would be:

```
I am       1936 342   90
I am       1945 211   10
very cool  1923 500   10
very cool  1980 3210  1000
very cool  2012 9994  3020
```

This tells us that, in 1936, the bigram ‘I am’ appeared 342 times in 90 different books. In 1945, ‘I am’ appeared 211 times in 10 different books. And so on.

Goal

For each unique bigram, compute its average number of appearances per book. For the above example, the results will be:

```
I am       (342 + 211) / (90 + 10) = 5.53
very cool  (500 + 3210 + 9994) / (10 + 1000 + 3020) = 3.40049628
```

Output the 10 bigrams having the highest **average number of appearances per book** along with their corresponding averages, in **tab-separated format**, sorted in descending order. If multiple bigrams have the same average, **order them alphabetically**. For the example above, the output will be:

```
I am       5.53
very cool  3.40049628
```

You will solve this problem by writing a PIG script on Amazon EC2 and save the output.

You can use the interactive PIG shell provided by EMR to perform this task from the command line (grunt). In this case, you can copy the commands you used for this task into a single file to have the PIG script and the output from the command line into a separate file. Please see this for how to use PIG shell. Also, you can upload the script and create a task on your cluster.

To load the data from the s3://cse6242-bigrams-small bucket into a PIG table, you can use the following command:

grunt> bigrams = LOAD ‘s3://cse6242-bigrams-small/*’ AS (bigram:chararray, year:int, count:int, books:int);
(**HINT**: You might want to change the data type for **year**, **count** or **books**)

Notes:

- Copying the above commands directly from the PDF and pasting on console/script file may lead to script failures due to the stray characters and spaces from the PDF file
- Your script will fail if your output directory already exists. For instance, if you run a job with the output folder as s3://cse6242-output, the next job you run with the same output folder (s3://cse6242-output) will fail. Hence, please use a different folder for the output for every run

While working with the interactive shell (or otherwise), you should first test on a small subset of the data instead of the whole data (the whole data is over hundreds of GB). Once you believe your PIG commands are working as desired, you can use them on the complete data and ...wait... since it will take some time.

To help you evaluate the correctness of your output, we have uploaded the output for the small dataset on T-Square (hyperlink).

Deliverables:

- **pig.txt**: The PIG script for the task (using the larger data set).
- **output-big.txt**: Output **(tab-separated)** for the larger data set.

Note: Please follow formatting requirements for Task 3, as we would be using an autograder

[15pts] Task 4: Explore and Analyze data with Pandas

You will use Python and Pandas to explore and analyze the movielens data[4] set. Save the results of 4a, 4b and 4c in movies.txt. If you are not familiar with Pandas, you can follow this basic tutorial - pandas-seaborn.

a) [4 pts] Import data and compute basic statistics.

Download task4data.zip. The data that you download has a readme.txt file which has details about how the data is stored. Using pandas, load the data as dataframes and find the number of unique movies and number of unique users in the dataset.

Output format:

Number_of_unique_movies
Number_of_unique_users

[4] This is a subset of data taken from http://grouplens.org/datasets/movielens/

b) [3 pts] Get top 5 movies with largest number of ratings.

Using pandas, find the top 5 movies with the most number of ratings. Sort the output from most ratings to least ratings.

Output format:

Movie_name<tab>Number_of_ratings

c) [3 pts] Average User Age by Movie

Find the top 5 movies with the lowest average age of the people that rated the movies. Consider only the movies that have got atleast 100 ratings.

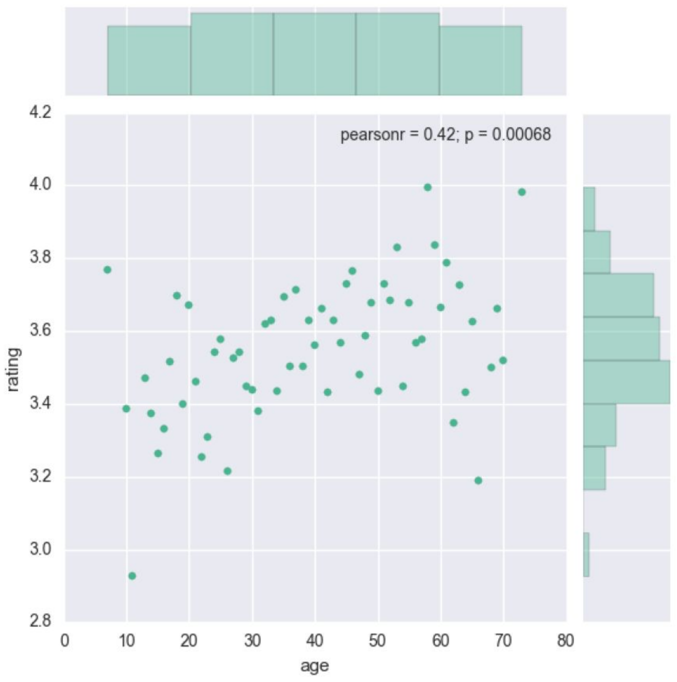
Output format:

Movie_name<tab>Number_of_ratings<tab>avg_age

d) [5 pts] Seaborn visualization

Draw a scatterplot with marginal histograms to visualize how the average rating varies with age. Save this plot as scatterhist.png.

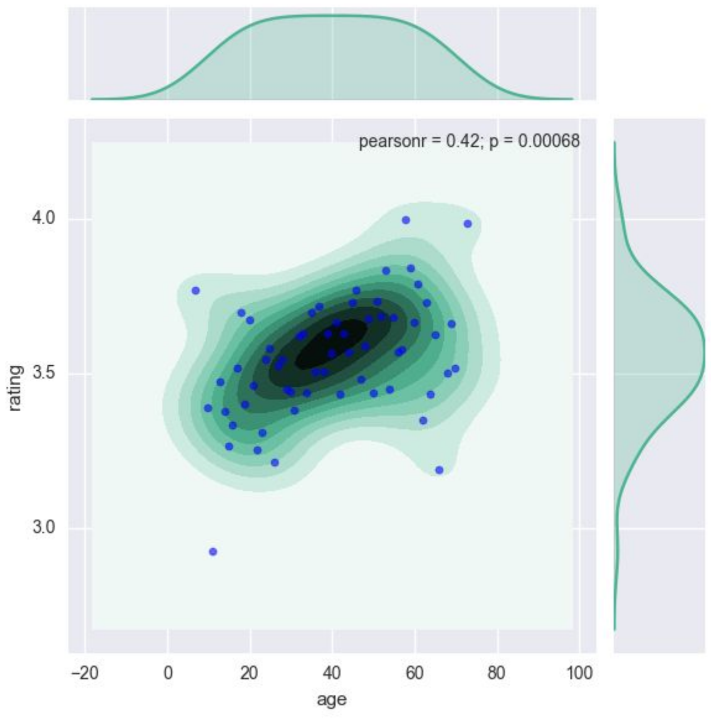
A sample plot is shown below (please note that it is just an example and may not be related to what you want to create. Also, it may be improved in a number of ways).



e) [Bonus 5 pts] Augment the scatterplot with density estimates

Please note that you will be awarded points for this section only if the graph has both the density estimates and the scatterplot on the same graph. Save this plot as scatterbonus.png.

A sample plot is shown below (again, it is just an example).



Deliverables:

- movies.py: The pandas script.
- movies.txt: The output of the script
- scatterhist.png: The plot for 4d.
- [Bonus] scatterbonus.png: The plot for 4e
- data: The folder which contains u.data, u.item, u.user and the readme.txt

Note 1: Do not include task4data.zip in the folder.
Note 2: Always use relative paths while loading the data as dataframes.

Submission Guideline

Submit the deliverables as a single **zip** file named **HW3-*Lastname-Firstname.zip***. Please specify the name(s) of any students you have collaborated with on this assignment, using the text box on the T-Square submission page.

The directory structure of the zip file should be exactly as below (the unzipped file should look like this):

```
HW3-Smith-John/
    Task1/
        src/main/java/edu/gatech/cse6242/Task1.java
        description.pdf
        pom.xml
        run1.sh
        run2.sh
        task1output1.tsv
        task1output2.tsv
        (do not attach target directory)
    Task2/
        src/main/scala/edu/gatech/cse6242/Task2.scala
        description.txt
        pom.xml
        run1.sh
        run2.sh
        task2output1.tsv
        task2output2.tsv
        (do not attach target directory)
    Task3/
        pig.txt
        output-big.txt
    Task4/
       movies.py
       movies.txt
       scatterhist.png
       scatterbonus.png [Bonus]
       data/
           readme.txt
           u.data
           u.item
           u.user
       (do not attach task4data.zip)
```

Please adhere to the naming convention specified above.