Big Data Analytics (SOEN 498/691)
Laboratory sessions

Tristan Glatard
Department of Computer Science and Software Engineering
Concordia University, Montreal
tristan.glatard@concordia.ca

February 8, 2017

Contents

1 Introduction **3**

I People You May Know **3**

2 Goal **3**

3 Basic idea **3**

4 First implementation **4**

5 Filtering existing friends **5**

II Who To Follow **7**

6 Goal **7**

7 Basic idea **7**

8 Assignment **8**

1 Introduction

Now that you are able to run a simple MapReduce example using Hadoop, we will focus on more complex algorithms. We will look at two popular applications used in social networks: "People You May Know" (Part I below) and "Who To Follow" (Part II). "People You May Know" is not an assignment. Links to solutions implemented in Java are provided but it is strongly recommended that you try to implement your own code before looking at the solutions. "Who To Follow" is an assignment due for the date indicated on the course outline. It is strongly recommended to complete "People You May Know" before doing "Who To Follow" as the solutions to these problems have a lot in common.

Part I
People You May Know

2 Goal

We will implement a basic "People You May Know" algorithm using MapReduce. We assume an un-directed social network of n users with a symmetrical "friend" relation among users, that is, if user `A` is a friend of user `B` then user `B` is also a friend of user `A`. Facebook is an example of such a network. Twitter is a counter example because user `A` may follow user `B` whereas user `B` does not follow user `A`. The algorithm will recommended to user `X` a list of friends $\texttt{Y}_i$ based on the number of friends that `X` and $\texttt{Y}_i$ have in common. The input and output of the algorithm will be as follows:

- Input: a file containing n lines with space-separated integers:
  `X` $\texttt{F}_1$ $\texttt{F}_2$ `...`
  where the $\texttt{F}_i$ are the ids of the friends of user `X`. For instance:

  (Link to file)

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

- Output: a file containing n lines in the following format:
  `X` $\texttt{R}_1\texttt{(n}_1\texttt{)}$ $\texttt{R}_2\texttt{(n}_2\texttt{)}$ $\texttt{R}_3\texttt{(n}_3\texttt{)}$ `...`
  where $\texttt{R}_i$ are the ids of the friends recommended to user `X` and n_i is the number of friends in common between `X` and $\texttt{R}_i$. Recommended friends $\texttt{R}_i$ **must be** ordered by decreasing values of $\texttt{n}_i$ and they **must not** be friends of `X`. For instance:

  (Link to file)

```
1
2 5(2) 4(1)
3 4(1)
4 3(1) 5(1) 2(1)
5 2(2) 4(1)
```

The following Sections will guide you through a possible implementation. Feel free to follow your own thread if you have any idea how to implement it: you may find a better solution than the one proposed!

3 Basic idea

The basic idea of our initial implementation is the following:

- Map step:
  - Receive (_,v): `_, X` $\texttt{F}_1$ $\texttt{F}_2$ ... $\texttt{F}_k$
  - Emit (k,v): $\texttt{F}_i$,$\texttt{F}_j$ where $i \in [1,k]$, $j \in [1,k]$ and $i \neq j$.

  In other words, `(a,b)` *and* `(b,a)` are emitted every time a friend common to user `a` and user `b` is found. The following key-value pairs are emitted in response to the input example above (order may of course vary):

(Link to file)

```
(2,3)
(2,4)
(2,5)
(3,2)
(3,4)
(3,5)
(4,2)
(4,3)
(4,5)
(5,2)
(5,3)
(5,4)
(1,3)
(3,1)
(1,2)
(1,5)
(2,1)
(2,5)
(5,1)
(5,2)
(1,3)
(3,1)
```

- Reduce step:
  - Receive (k, [v]): `X, [` $\texttt{F}_1$, $\texttt{F}_2$ ... `]`
    In this list the $\texttt{F}_i$ are not unique: instead, $\texttt{F}_i$ appears exactly `x` times when user `X` and user $\texttt{F}_i$ have `x` friends in common. For instance, the following key-value pairs are received in our example:

    (Link to file)

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

  - Emit (k, v): `X, "`$\texttt{F}_1$`(`$\texttt{n}_1$`)` $\texttt{F}_2$`(`$\texttt{n}_2$`)` ..."
    where $\texttt{F}_i$ appeared exactly $\texttt{n}_i$ times in the received values. In our example:

    (Link to file)

    ```
    1 3(2) 5(1) 2(1)
    2 5(2) 3(1) 4(1) 1(1)
    3 1(2) 2(1) 5(1) 4(1)
    4 5(1) 3(1) 2(1)
    5 2(2) 4(1) 1(1) 3(1)
    ```

4 First implementation

Implement the algorithm described in the previous Section. Make sure that it produces the correct result on the example above. Then, test your implementation on larger networks by generating input files using a program such as `generate.py`.

Here is a possible solution:

(Link to file)

5 Filtering existing friends

There is a problem with the previous implementation as it may recommend to user `X` people that are already friend with `X`. For instance, in the example above, user `3` is recommended to user `1` although `1` and `3` are already friends. We need to somehow emit key-value pairs containing the friends of user `i`, to address this issue. Here is a possible solution:

- Map step:
  - Receive (_,v): *no change*.
  - Emit (k,v):
    - $\texttt{F}_i$,$\texttt{F}_j$ where $i \in [1, k]$, $j \in [1, k]$ and $i \neq j$.
    - `X`, `-`$\texttt{F}_i$ for all $i \in [1, k]$ (note the `-` sign).

    For instance, in the example above:

    (Link to file)

```
(2,3)
(2,4)
(2,5)
(3,2)
(3,4)
(3,5)
(4,2)
(4,3)
(4,5)
(5,2)
(5,3)
(5,4)
(1,-2)
(1,-3)
(1,-4)
(1,-5)
(1,3)
(3,1)
(2,-1)
(2,-3)
(1,2)
(1,5)
(2,1)
(2,5)
(5,1)
(5,2)
(3,-1)
(3,-2)
(3,-5)
(1,3)
(3,1)
(5,-1)
(5,-3)
```

- Reduce step:
  - Receive (k, [v]): `X, [` $\texttt{F}_1$`,` $\texttt{F}_2$`, ... ]`

If F_i is negative then it is a friend of user X and all occurrences of -F_i in the values must be removed. Otherwise, F_i is a user who has friends in common with X and its number of occurrences must be counted as before. In our example:

(Link to file)

```
( 1, [ 2 , 3 , 3 , 5 , -2 , -3 , -4 , -5 ] )
( 2, [ 1 , 3 , 4 , 5 , 5 , -1 , -3 ] )
( 3, [ 1 , 1 , 2 , 4 , 5 , -1 , -2 , -5 ] )
( 4, [ 2 , 3 , 5 , -1 ] )
( 5, [ 1 , 2 , 2 , 3 , 4 , -1 , -3 ] )
```

– Emit (k, v): X, "F_1(n_1) F_2(n_2) ..."
where F_i appeared exactly n_i times in the received values after the friends of X have been removed. In our example:

(Link to file)

```
1
2 5(2) 4(1)
3 4(1)
4 3(1) 5(1) 2(1)
5 2(2) 4(1)
```

Implement this solution and verify that it works correctly.
Here is a possible solution:

(Link to file)

Part II
Who To Follow

6 Goal

We now assume a directed social network of n users with an asymmetrical "follows" relation among users, that is, user B doesn't necessarily follows user `A` even though user `A` follows user `B`. Twitter is an example of such a network. The goal is to write an algorithm that will recommend to user `X` a list of people to follow $\texttt{F}_i$ based on the number of followers that `X` and $\texttt{F}_i$ have in common. The input and output of the algorithm will be as follows:

- Input: a file containing n lines with space-separated integers:
  $\texttt{X F}_1\ \texttt{F}_2\ \ldots$
  where $\texttt{F}_i$ are followed by user `X`. For instance:

  (Link to file)

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

- Output: A file containing n lines in the following format:
  $\texttt{X R}_1\texttt{(n}_1\texttt{)}\ \texttt{R}_2\texttt{(n}_2\texttt{)}\ \texttt{R}_3\texttt{(n}_3\texttt{)}\ \ldots$
  where $\texttt{R}_i$ are the ids of the people recommended to user `X` and n_i is the number of followed people in common between `X` and $\texttt{R}_i$. Recommended people $\texttt{R}_i$ **must not** be followed by `X` and **must** be ordered by decreasing values of $\texttt{n}_i$. On the previous example:

  (Link to file)

```
1 2(2)
2 4(3)
3
4
5 2(1) 1(1) 4(1)
```

7 Basic idea

Similarly to what we did for "People You May Know", you may want to write a mapper that emits (`a`,`b`) and (`b`,`a`) every time user `a` and user `b` follow someone in common. However, because of the fact that the "follows" relation is asymmetrical, this requires two MapReduce jobs:

1. Indexing: for each user $\texttt{F}_i$ followed by user `X`, the mapper emits $\texttt{F}_i$ as the key and `X` as the value. The reducer is the identity. It produces inverted lists of followers:
   $\texttt{X}, [\ \texttt{Y}_1, \texttt{Y}_2, \ldots, \texttt{Y}_k\ ]$
   where the $\texttt{Y}_i$ all follow user `X`.

2. Similarity: for each inverted list $\texttt{X}, [\ \texttt{Y}_1, \texttt{Y}_2, \ldots, \texttt{Y}_k\ ]$ the mapper emits all pairs ($\texttt{Y}_i$,$\texttt{Y}_j$) and ($\texttt{Y}_j$,$\texttt{Y}_i$) where $i \in [1, k]$, $j \in [1, k]$ and $i \neq j$. As in "People You May Know", the reducer receives a list $\texttt{X}, [\ \texttt{F}_1, \texttt{F}_2, \ldots\ ]$ where $\texttt{F}_i$ appears exactly `x` times if `X` and $\texttt{F}_i$ follow `x` people in common. It counts the occurrences of $\texttt{F}_i$ whenever $\texttt{F}_i$ is not followed by `X` and sorts the resulting recommendations by number of common followed people.

To filter the list of suggested people from followed people, you can use the same kind of trick as for "People You May Know", using negative integers.

8 Assignment

Write a MapReduce program that implements a basic "Who To Follow" system fulfilling the requirements described in Section 6. You may:

- Follow the idea in 7 or your own idea.
- Use Java or any other programming language (using Hadoop Streaming).

You must:

- Make sure that your program will work on Linux.
- Include a README file explaining how to compile (if relevant) your program, how to run it and where to find the results. Mention all the requirements, e.g., the version of Java required.
- Use Git to track your developments. Your commit history will be checked and frequent commits showing incremental development will be favored over monolithic ones.

How to submit your assignment:

- Your assignment must be submitted on Moodle, as a zip, tar or tar.gz archive. If you are submitting in a team of two, each team member must submit a copy of the assignment.
- Your archive must contain:
  - a README file containing your name(s) and student id(s) and explaining how to compile (if relevant) your program, how to run it and where to find the results. Mention all the potential requirements, e.g., the version of Java required.
  - the source code of your program (.java, .py, etc).
  - a compiled version of your program (if relevant), for instance a jar file containing all the classes.
  - a .git directory containing your git history (this directory is created by git when you initalize a git repository).

You don't have to submit a pdf file for your assignment.

Your program will be tested on the example data of Section 6 as well as on undisclosed evaluation data generated with `generate.py`. Marking will be as follows:

- 10%: Program compiles (if relevant) and MapReduce job(s) run to completion.
  - Program doesn't compile or crashes: 0%.
  - Program compiles and completes successfully: 10%.
- 20%: MapReduce job(s) produce correct output on Section 6's data.
  - Recommendations are incorrect: 0%.
  - Recommendations contain followed people or are not correctly sorted: 10%.
  - Output is correct: 20%.
- 50%: MapReduce job(s) produce correct output on evaluation data (undisclosed).
  - Recommendations are incorrect: 0%.
  - Recommendations contain followed people or are not correctly sorted: 25%.

  - Output is correct: 50%.

- 10%: Program is easy to read and understand.
  - It takes more than 15 minutes to understand: 0%.
  - It takes more than 10 minutes to understand: 5%.
  - It takes less than 10 minutes to understand: 10%.
- 10%: Git commit history is available and shows frequent commits.
  - Commit history is not available: 0%.
  - Commit history is available but shows only a few large commits: 5%.
  - Commit history is available and demonstrates frequent commits showing incremental development steps: 10%.