Lab 3: Network Prediction and Sampling

March 5th, 2014

Objectives

After looking at structural properties during Lab 2, this one will focus on two additional problems related to networks. The first one is *link prediction*: Given a snapshot of a network, can we infer future interactions between its members? Can we use the current structure of the network to predict its evolution? Which accuracy can we reach?

The second problem is *graph sampling*. In order to obtain a representative sample of members of a network, a possible approach is to crawl it. At first, this method seems seducing. However, we will show, through a realistic scenario, how such a simplistic approach can lead to significant estimation errors.

As in the previous labs, you will find the relevant files in a tar archive, `ix-lab03.tar.gz`, available on Moodle.

Deliverables

Before the start of the next lab, you should hand in (through Moodle) a short report with the following information.

- The code of your implementation of the class `MyOwnScoring`
- The accuracy of the three link prediction strategies of Section 1.
- Short answers to the questions asked throughout the handout. These will serve as a basis for small interviews that we will conduct later in the semester.

1 Link Prediction

You are a Ph.D student in the Internet Analytics department of a prestigious university. You love to analyze data and uncover its hidden patterns. Billy, a talent agent whose job is to find the stars of the future, contacts you: he believes that the future of Hollywood lies within the data and that crunching large amounts of data is the best way to discover new talented actors. Since he has no expertise in computer science, he wants you to write an algorithm to predict exclusive future collaborations between actors. He believes that the ability to foresee future collaborations would give him a significant advantage over his competitors.

As you love a good challenge, you accept the job and decide to write a co-actor prediction algorithm. You start by crawling the Internet Movie Database (IMDb) website to gather data about actors and movies. You model the co-appearance between actors as a graph where a vertex represents an actor, and an edge between two actors means that they co-appeared in at least one movie.

1.1 The Dataset

We will build the network of actor co-appearance from on a dataset released by the Internet Movie Database[1]:

- On HDFS, the file `/ix/imdb.txt` contains the entire dataset.
- In the `data/` folder of the archive you extracted on your computer, you will find a small sample in the file `imdb-sample.txt`.

These files contain lines of the form `[actor name],[movie title]`. To distinguish between different actors bearing the same name, a number is sometimes appended. For all practical purposes, you can simply consider the numbers as being part of the name, and the names as being unique. Here is a small excerpt of the dataset:

```
De Niro Robert,Mistress (1992)
Fabre Saturnin,Les portes de la nuit (1946)
Ayala Vicente,La isla interior (2009)
Caine Roger,While the Cat's Away... (1972)
Greco Sam,Under the Gun (1995)
Tsobe Geoffrey,Diamond Safari (1958)
Stoyannidis Vassilis,Metaihmio (1994)
Moreno Antonio (I),Ambush (1939)
James Joe (III),Analyze That (2002)
Moore Ida,Girls of the Big House (1945)
```

We provide you with a class, `ix.utils.IMDbEntry`, that can be used to parse a line of the dataset and extract actor name, movie title and year of release.

1.2 Extracting Movie Casts

The first step will be to extract the movie casts, i.e., group all the actors that have participated in a movie together. Using a MapReduce job, you will parse the IMDb dataset and output the cast of all movies in the following format:

```
movie title        actor name 1, actor name 2, ...
```

Complete the Mapper and the Reducer, `MovieCastMapper` and `MovieCastReducer`, in the package `ix.lab03.extraction`.

1. Test your implementation using the provided unit tests.
2. Run `MovieCast` locally on the sample dataset, `imdb-sample.txt`.

[1] IMDb, see: `http://www.imdb.com/`.

3. Once everything works well, export a JAR archive, transfer it to a cluster access server, and run the job:

```
hadoop jar ix-lab03.jar ix.lab03.extraction.MovieCast /ix/imdb.txt lab3/moviecast
```

What is the movie with the largest cast? Use the script `scripts/largest-cast.sh` on the output[2] of the MapReduce job to find it. A famous French actor played in this movie. Who is he?

1.3 Co-Actor Graph

Building on the cast of the movies you have just obtained, use the MapReduce model to construct the co-appearance graph. We will store this graph as an adjacency list[3], i.e., for each actor, we list the actors with whom he or she acted at least once. The output should have the following format:

```
actor name        co-actor name 1, co-actor name 2, ...
```

With respect to all the MapReduce jobs that we introduced so far, this one has a small twist. We would like to filter the movies we take into consideration, and ignore the ones that were not produced in a certain timeframe. This information has to be passed on to all the mappers, and this can be done through the job's `Configuration` object.

Complete the Mapper and the Reducer, `CoActorMapper` and `CoActorReducer`, and make sure that your mapper respects the starting year and ending year thresholds.

Just like in the previous step, you can test your implementation using the unit tests, run the code locally, and export your code to run it on the cluster. Do not forget that this second job takes the output of the first job as input data. We also provide you with a job, `IMDbCoActorGraph`, that wraps the movie cast job and the co-actor graph one in a single command. For the starting and ending year, you can choose the period of interest to be e.g. from 2000 to 2008.

1.4 Helping Billy Make Money

Billy contacts you to have an update on the status of your work. He is impatient to have the first results about future collaborations.

In our setup, predicting future co-appearances of actors boils down to the well-known link-prediction problem. We will implement three link prediction algorithms. They all operate on the same principle; they consider every possible new edge and assign a score such that the higher the score, the more likely the edge will be formed. Once a score has been assigned to every potential edge, we will keep the top N edges. The scoring functions are as follows.

- Preferential attachment: the score of an edge is assigned according to the "rich get richer" phenomenon. See your class notes for more details.
- Number of common neighbors: here the idea is to give a high score to pairs of nodes who are well connected through their neighborhood.
- The third scoring function is completely up to you. Can you design a measure that beats the two other scoring functions in terms of prediction accuracy?

The algorithms will be evaluated based on two snapshots of the IMDb co-actor graph built using the jobs described aboove. As we did some additional pre-processing to filter out amateur movies and restrict the number of actors to a tractable amount, we provide you with the files directly.

- The file `data/snapshot-2006.txt` contains the co-actor graph obtained when considering movies up to 2006. This will be the data that you use to assign a score to new, not-yet-existing edges.

[2] Do not forget to copy the output of the job from HDFS to your local file system using `getmerge`.

[3] Adjacency lists are one of several ways of representing a graph. In Lab 2, for the graph of links between Wikipedia articles, we represented the graph differently, as a *list of edges*.

- The file `data/snapshot-2012.txt` contains a similar graph, but this time with movies up to 2012. Only actors that were present in 2006 are retained, meaning that the nodes are the same. However, many new edges have been formed, and these new edges will serve as a *ground truth* against which your link-prediction algorithms will be compared.

Successively complete the classes `CommonNeighbors`, `PreferentialAttachment` and `MyOwnScoring` inside of `CoActorPrediction` (in the package `ix.lab03.prediction`.) Run `CoActorPrediction` with the two co-actor graph snapshots, and compare the predictive accuracy of the three different algorithms.

Note: formally, we have two graphs, $G_1(V, E_1)$ and $G_2(V, E_2)$, where $E_2 = E_1 \cup E_{\text{new}}$. Let us denote by $N = |E_{\text{new}}|$ the number of new edges in G_2. A predictor p will assign a score to each of the $\binom{|V|}{2} - |E_1|$ edges that are not in G_1, and keep the N edges which have the highest score into a set E_p. We would then like to measure how close E_p is from E_{new}. To do this, we look at the relative size of the intersection of these two sets.

$$\text{accuracy} = \frac{|E_p \cup E_{\text{new}}|}{|E_{\text{new}}|}.$$

This quantity is also called the *recall*, and is always between 0 (no edges were predicted correctly) and 1 (all the edges were predicted correctly.)

Since your thesis rotates around random graphs, you decide to apply your amazing algorithm on a instance of a random graph $G(n, p)$. You generate a $G(n, p)$, and you remove uniformly at random some edges. What would be the performance of your algorithm on predicting these missing edges? What is the best strategy for finding these missing links? Justify rigorously your answer (and think before coding ¨⌣)

2 Random Walks on Social Networks

You are curious about the age distribution of the users of `Facebook`. As you cannot have this information directly from `Facebook` itself, you decide to write a small program that allows you to crawl the public `Facebook` network. You start from your profile and proceed iteratively as follows:

Algorithm 1 Random walker on `Facebook`

Require: N
  $x \leftarrow$ u
  $i = 0$
  while $i < N$ **do**
    Get the age the node x
    Select node v uniformly at random from the neighbourhood of x
    $x \leftarrow v$
    $i \leftarrow i + 1$
  end while

Facebook provides a way to collect information about a node in the network through HTTP requests[4]. We provide you with a Java Application Programming Interface (API), the class `SocialAPI`, that wraps these requests and provides a convenient way to navigate on the network. In particular, this class provides the method `getNode()` that takes the ID of a node (say, x) and returns you a `SocialNode` object with information on

[4]While the *real* Facebook does provide HTTP APIs, the one we are using for this lab is fictitious and the underlying social graph was created from scratch.

- the corresponding user's age,
- the corresponding user's list of friends.

In the following, we assume that the network is connected and remains static for the time of the experiment. Write your code in the class `FacebookSampling`.

1. Setting $N = 10\,000$ and node u as initial seed[5], what is the average age of a `Facebook` user?
2. A study whose title is "Facebook, by Facebook" is released, and it reveals that the average age of `Facebook`'s users is 52.6 years old. How close is you estimation to the true average age? How do you explain it?
3. What would you change in your crawler in order to have a more accurate estimate of average age? You solution should require neither additional seeds nor more memory allocated.

Note: we strongly encourage you to display the running average after each node, in order to see how it evolves over time. This makes debugging easier, and might help you get some additional insights.

2.1 Bizarre Social Networks

Now you want to use your algorithm to crawl a new and trendy social network called `Ages` (whose average user is 60 years old). Write your code in the class `AgesSampling`.

1. Setting $N = 10\,000$, what is the average age of `Ages`'s users when you start your crawl from node u? from node v? from node w? Can you explain which properties of the network could cause this?
2. What would you change in your crawler in order to have a more accurate estimate of average age? *Hint:* Starting your crawler from node u, increase N and observe the impact on the age estimate.

Finally, you want to crawl yet another social network called `Directions` (whose average user is, we heard, 32.5 years old.) Complete the code in `DirectionsSampling`. What estimate of the average age of `Directions`'s users do you get when you start your crawl from node u? From node v? From node w? Explain what you obtain.

[5]Please refer to the code in order to have the identifiers of the seed nodes u, v and w.