Algorithms for Massive Data project: Using PageRank and Topic Sensitive PageRank for Link Analysis

Doina Vasilev - 00080A

December 2023

1 Introduction

The following report offers an in-depth analysis of my project, where the Link Analysis methodologies, specifically the Page Rank and Topic Sensitive Page Rank algorithms, were applied comprehensively to the *Amazon US Customer Review* dataset, published on Kaggle under the amazon.com conditions of use. Two products will be linked if they have been reviewed at least by the same customer. The dataset contains several csv files, each containing information on a specific product category. For the purposes of the project, I've chosen the file on *books* sold on Amazon.

2 Link Analysis and Page Rank

Link Analysis is a key aspect in determining the importance and relevance of a web page in the context of Page Rank. The idea that the value of a web page can be determined by the pages that link to it is fundamental to almost all web search tools. This is based on the premise that the more pages that link to a page, the more important that page is considered to be.
However, this definition appears to be useless, since it's a common situation of recursive with no base case scenario.
We can start by envisioning the Web as a directed graph, where the pages are the nodes of the graph and the edges are the hyperlinks connecting the pages, like in Fig.1. At time $t = 0$, there are n random surfers, who are uniformly assigned to the nodes of the graph. In the next step, each one of them can only move from one node to the other if there's a link connecting the first node (j), to the second (i). We can define as *transition matrix*, the column-wise stochastic matrix having on the rows the *in-going links* and on the columns the *out-going links*. This matrix M has n rows and columns, if there are n pages. The element m_{ij} in row i and column j has value *1/k* if page j has k arcs out, and one of them

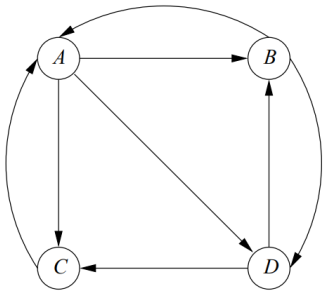

Figure 1: The Web represented as a directed graph.

is to page i. Otherwise, $m_{ij} = 0$.
The probability distribution of the location of a random surfer can be described by a column vector whose jth component is the probability that the surfer is at page j. This probability is the (idealized) PageRank function.
As previously mentioned, surfers are randomly, and uniformly, distributed across the nodes of the directed graph. Therefore, the initial vector v_0 will have $1/n$ for each component.
At the next step, $v_1 = Mv_0$ where M is the transition matrix and v_0 is the vector describing the probability distribution across the nodes at $t = 0$. Iterating the same multiplication, the distributions eventually converge to a vector $\mathbf{v}$. This approach is called *Power Method*.
The vector $\mathbf{v}$ will provide the distribution of the random surfers across the Web, which we can interpret as the *Page Rank* of each page on the Web.
To find the distribution of the surfer $\mathbf{v}$ that satisfies $\mathbf{v} = Mv$, two conditions are to be met:

1. The graph is strongly connected; that is, it is possible to get from any node to any other node.

2. There are no dead ends: nodes that have no arcs out.

Under these conditions, we can rest assured that the limit is reached when multiplying the distribution by M another time does not change the distributions.
This is due to the fact that the limiting $\mathbf{v}$ is an eigenvector of M, as the *eigenvector* of a matrix M is a vector $\mathbf{v}$ that satisfies $\mathbf{v} = \lambda M\mathbf{v}$ for some constant *eigenvalue* λ. Not only that, but $\mathbf{v}$ is the *principal* eigenvector of M.

3 Topic Sensitive Page Rank

There are several improvements we can make to PageRank. One, to be studied in this section, is that we can weight certain pages more heavily based on their topic. The mechanism for enforcing this weighting is to alter the way random surfers behave, having them prefer to land on a page that is known to cover the chosen topic.
The idea behind Topic-Sensitive Page Rank is that different people have different interests, which can alter their choice on the next step of the graph.
Therefore, to take this aspect into account, we shall consider three more elements to add to the previous computation of Page Rank:

1. S, the set of integers consisting of the row/column numbers for the pages we have identified as belonging to a certain topic. Also called *teleport set*.

2. e_S, the vector that has 1 in the components in S and 0 in all other components.

3. β, the coefficient representing the probability that the surfer will follow the path of transition matrix or will be teleported to another page on the graph. Usually it's set around 80%.

Then, the *Topic-Sensitive Page Rank* is computed as follows:

$$v' = \beta * Mv + (1 - \beta)\frac{e_S}{|S|} \tag{1}$$

As the name says, Topic - Sensitive Page Rank assigns larger weight to the pages whose topic surfers are most interested in.
As we will see, instead of applying a real Topic - Sensitive Page Rank, I have rather applied a **Rating - Sensitive** Page Rank, where the ranking is based on the *average ratings* of the books. However, the mathematical foundations and concepts are the same.

4 Data

The dataset used for this project is retrieved from Amazon US Customer Review, published on Kaggle under the amazon.com conditions of use. In particular, I've decided to work on *amazon_reviews_us_Books*, Amazon's books' reviews dataset.
Due to the size of the data, the traditional model to deal with them would be computationally expensive and time-consuming. Therefore, resilient distributed datasets (or RDDs) are used.
Resilient Distributed Datasets (RDDs) are a fundamental data structure of Spark. It is an immutable distributed collection of objects, where each RDD is divided into partitions, which may be computed on different nodes of the cluster.
Thanks to parallel processing, RDDs ensure a fast and efficient processing of large amounts of data, in addition to several other desirable characteristics, such as *data resilience*, *data consistency* and *performance speed*.

5 Experiments

5.1 Implementation of Page Rank

The following code performs a link analysis on Amazon's book reviews. Due to the size of the dataset, I'll be using **PySpark** package, which allows for distributed data processing. This provides the ability to handle large datasets by utilizing Spark's capability for in-memory and on-disk storage.

In the first part of the project I've only implemented a baseline version of Page Rank, to determine which books attract the most Amazon's readers.
In this case, books are linked if they have been reviewed by the same costumers.
Before implementing the algorithm, I've performed some data pre-processing:

1. The header gets eliminated, after which the data gets split into two columns, respectively containing costumer and product ids. Due to limitations in CPU, I've sampled 10% of the dataset, however, thanks to the use of RDDs, the analysis can be conducted on larger portions of it.

2. Then, the data is grouped by *costumer_id*, and each costumer is mapped to the list of books they have reviewed. In the following, I'll be processing approximately 310000 data points.

3. Finally, to avoid *dead ends*, customers who have reviewed only one book are filtered out of the dataset.

Now it's time to define the variables that I'll be using for computing the Page Rank.
First, I compute the total number of nodes, i.e., the total number of distinct books in the dataset:

```
# Compute the total number of nodes (distinct book id)
tot = data.map(lambda x: x[1]).distinct().count()
```

The total number of nodes will be useful later, when I'll be initializing the vector representing the distribution of the 'random' readers at step 0.

Secondly, I've defined a function that will return all possible pairs (edges) between books (nodes) in the input data.

In the function, the variable **combine** creates a list of pair values. It starts by taking the first element (*v1*) and pairing it with all the following elements in the list (*v2*, which are fetched from the subset of *values* list that starts from index i+1). Then it proceeds to the second element and pairs it with all the following elements and so on.
Then, **add** creates a new list where all pairs from the *combine* list are in the reverse order. Finally, the function returns a list that combines the two lists.

```
# Compute the edges between books
def calculate_linkages(data):
    key, values = data
    combine = [(v1, v2) for i, v1 in enumerate(values) for v2 in values[i + 1:]]
    add = [(v2, v1) for (v1, v2) in combine]
    return (combine + add)
```

The next step is to define the most relevant variables for creating a *transition matrix*. First, **links** creates a list of tuples, each containing pairs of connected books. Then, **id2degree** computes the number of total edges for each book.

```
# Compute the list of edges
links = filtered.map(lambda x: calculate_linkages(x)).flatMap(lambda value: value)

# Calculate out-degree for each node
id2degree = links.countByKey()
```

Finally, I'm able to compute the transportation matrix: first, I create **P**, as the matrix having the out-going nodes on the rows, and the in-going nodes on the columns. Then, for each pair, the elements of the matrix are determined by the number of outgoing links from each node *i*. Then, I map the matrix into the transposed, having the in-going nodes on the rows and the out-going ones on the columns. This ensures the creation of a *column-wise stochastic matrix*.

```
# Create the transportation matrix and its transposed
P = links.map(lambda x: (x[0], x[1], 1 / id2degree[x[0]])) #(i, j, mij)
PT = P.map(lambda x: (x[1], x[0], x[2])) #(j, i , mij)
```

The next step requires to create a function, that computes the initial probability for each node as $1/total$, where *total* was previously defined as the total number of books in the set.

```
# Calculate the initial probability
def calculate_probability(degrees, total):
    prob = 1 / total
    p_i = {item: prob for item in degrees.keys()}
    return p_i

p_i = calculate_probability(id2degree, tot)
```

Finally, I can exploit the **Power Method** for Page Rank, by iteratively updating the probabilities. The Power Method takes the transition matrix, and iteratively updates the probabilities for each book. The result is then stored in a sorted dictionary, which will be useful later.

```
# Power Method
for i in range(50):
    new_p = PT.map(lambda x:(x[0], (x[2]*p_i[x[1]])))\
              .reduceByKey(lambda x,y: x+y)\
              .collect()\

    for idx,prb in new_p:
        p_i[idx] = prb

# Save the results into a dictionary
d = dict(new_p)

# Sort it by product in ascending order
sorted_d = dict(sorted(d.items(), key=lambda item: item[0], reverse=False))
```

The previous chunk of code was retrieved from assistant professor Bertolotti's tutorial.

5.2 Implementation of Topic-Sensitive Page Rank

As an improvement to the project, I've decided to implement a Topic-Sensitive Page Rank. However, due to the lack of information on the topics of the books (most reviews only mention how good or bad the book is), I've decided to introduce a small change to the concept. Indeed, I've made the decision of computing the Page Rank based on the **average rating** of the books in the input data, rather than computing it based on the topic of the books. In other words, I'm now implementing a **Rating - Sensitive** Page Rank, in which we envision two groups of readers: those who prefer *higher*-rated books, and those who tend towards *lower*-rated books.
With respect to the past implementation, I've made a small fix to the list of books associated to each costumer, as I've also introduced the rating that each costumer assigned to the respective book.
Then, I've created an RDD that only retrieves the pairs book and rating from the RDDs containing the filtered triples: costumer, book and rating.

```
# Group the data by costumer id and map values into a list of tuples
#(product_id, rating)
dfts = datats.groupBy(lambda x: x[0])\
    .mapValues(lambda values: [(value[1], value[2]) for value in values])
# Filter out dead ends
filteredts = dfts.filter(lambda x: len(x[1]) > 1)
# From the filtered df retrieve the list of tuples (product_id, rating)
S = filteredts.flatMap(lambda x: x[1])
```

Afterwards, I've created the **average_ratings** RDD, which returns the average rating for each book.

```
# Compute the average rating per book
# - Convert rating to int and create a count
# - Sum ratings and count for each product
# - Divide sum of ratings by count to get average
average_ratings = S
        .map(lambda x: (x[0], (int(x[1]), 1)))
        .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
        .mapValues(lambda x: int(x[0] / x[1]))
        .sortBy(lambda x: x[0])
```

The first *map* function maps each book_id to a tuple of rating and count.
Then, the *reduceByKey* function groups by book_id and computes the sum over the ratings and the counts, in order to obtain the sum of ratings and the total number of times in which the book is reviewed. Finally, for each key, *mapValues* computes the average rating, and sorts by key, to match the order of the elements in sorted Page Rank dictionary.
As I previously mentioned, I create two groups of people. The first group is only interested in books with a rating larger or equal than 4 (on a 1 to 5 rating system). The second group is only interested in books with lower ratings.
First, I filter out nodes having a rating smaller or equal to 4, and create the set **S_above3** of nodes containing only the books with average rating fitting the criterium. Then, I take the cardinality of the set S, in **S_count_above3**, and the topic-sensitive vector **e_S_above3**. In the vector, each of its elements is associated to one book, and each one of them is assigned value 1 if the book is rating-sensitive, or 0 otherwise.

```
# Filter out nodes having a rating greater or equal to 4
S_above3 = average_ratings.filter(lambda x: float(x[1]) >= 4)

# Cardinality of the set of topic-sensitive products
S_count_above3 = S_above3.count()

# Compute the topic-sensitive vector:
e_S_above3 = average_ratings\
              .map(lambda x: int(float(x[1])>= 4))\
              .collect()
```

The same computations are performed over the set of data with ratings lower than 4:

```
# Filter out nodes having a rating larger than 4
S_less4 = average_ratings.filter(lambda x: int(x[1]) < 4)

# Cardinality of the set of topic-sensitive products
S_count_less4 = S_less4.count()

# Compute the topic-sensitive vector:
e_S_less4 = average_ratings\
              .map(lambda x: int(int(x[1]) < 4))\
              .collect()
```

Finally, I create the function computing the Rating Sensitive Page Rank for each item. The function takes as input an RDD **d**, a vector **e**, and two parameters, **beta** and **n**, respectively a coefficient, usually around 0.8 and 0.9, and the cardinality of the set of topic-sensitive books. To showcase the different cases, I've implemented the algorithm twice, with beta = 0.8 and beta = 0.6.

```
# Compute the Topic Sensitive PR for each item
def TopicSensitivePR(d, e, n):
    TSPR = {}
    beta = 0.8 #0.6
    for (key_pr, pagerank), element in zip(d.items(), e):
      new_value = 0
      new_value += beta * pagerank + (1 - beta) * element / n
      TSPR[key_pr] = new_value
    return TSPR
```

6 Results

In this final section, I'll be showcasing the results obtained in the previous section.
First, we can observe the results of **Page Rank** by printing the 10 most reviewed books, and their respective page rank:

```
Most quoted products:
With prob: 0.00011288455081260919, you take product with code: 0373250517
With prob: 0.0001063244794487078, you take product with code: 0345458931
With prob: 0.00010014309221280358, you take product with code: 0399152180
With prob: 8.79062811490297e-05, you take product with code: 0385504209
With prob: 8.46592070356197e-05, you take product with code: 0385336675
With prob: 8.234647965291267e-05, you take product with code: 0385333927
With prob: 8.138651979442579e-05, you take product with code: 043935806X
With prob: 8.043795194239667e-05, you take product with code: 0767908171
With prob: 7.940514697075e-05, you take product with code: 0312252617
With prob: 7.871454788775515e-05, you take product with code: 0451524934
```

Based on the random 10% sample of the dataset, these are the 10 most reviewed books:

```
[('0373250517', 'Spitting Feathers (Red Dress Ink)')]
[('0345458931', 'Body Double')]
[('0399152180', 'Melancholy Baby (A Sunny Randall Novel)')]
[('0385504209', 'The Da Vinci Code')]
[('0385336675', 'The Enemy (Jack Reacher, No. 8)')]
[('0385333927', 'Pagan Babies')]
[('043935806X', 'Harry Potter and the Order of the Phoenix (Book 5)')]
[('0767908171', 'A Short History of Nearly Everything')]
[('0312252617', 'Fast Women')]
[('0451524934', '1984 (Signet Classics)')]
[('0375826688', 'Eragon (Inheritance)')]
```

Now, we can observe how the results change as soon as we introduce the **ratings** in the computation.
With beta = 0.8, we obtain the following results:

1. Topic-Sensitive PR, when we select the books with average ratings larger or equal to 4:

   ```
   [('0373250517', ('Spitting Feathers (Red Dress Ink)', 5.0))]
   [('0345458931', ('Body Double', 4.167))]
   [('0399152180', ('Melancholy Baby (A Sunny Randall Novel)', 4.0))]
   [('0385336675', ('The Enemy (Jack Reacher, No. 8)', 4.429))]
   [('0385504209', ('The Da Vinci Code', 3.841))]
   [('043935806X', ('Harry Potter and the Order of the Phoenix (Book 5)', 4.464))]
   [('0767908171', ('A Short History of Nearly Everything', 4.706))]
   [('0312252617', ('Fast Women', 4.6))]
   [('0451524934', ('1984 (Signet Classics)', 4.794))]
   [('0385333927', ('Pagan Babies', 3.625))]
   ```

2. Topic-Sensitive PR, when we select the books with average ratings smaller than 4:

   ```
   [('0373250517', ('Spitting Feathers (Red Dress Ink)', 5.0))]
   [('0345458931', ('Body Double', 4.167))]
   [('0385504209', ('The Da Vinci Code', 3.841))]
   [('0399152180', ('Melancholy Baby (A Sunny Randall Novel)', 4.0))]
   [('0385333927', ('Pagan Babies', 3.625))]
   ```

```
[('0375826688', ('Eragon (Inheritance)', 3.708))]
[('039914563X', ('The Bear and the Dragon', 2.2))]
[('0871138646', ('Old Flames', 3.5))]
[('0385336675', ('The Enemy (Jack Reacher, No. 8)', 4.429))]
[('0312873441', ('Running On Instinct', 3.5))]
```

With beta = 0.6, instead:

1. Topic-Sensitive PR, when we select the books with average ratings larger or equal to 4:

```
[('0373250517', ('Spitting Feathers (Red Dress Ink)', 5.0))]
[('0345458931', ('Body Double', 4.167))]
[('0385504209', ('The Da Vinci Code', 3.841))]
[('0399152180', ('Melancholy Baby (A Sunny Randall Novel)', 4.0))]
[('0385333927', ('Pagan Babies', 3.625))]
[('0375826688', ('Eragon (Inheritance)', 3.708))]
[('039914563X', ('The Bear and the Dragon', 2.2))]
[('0871138646', ('Old Flames', 3.5))]
[('0385336675', ('The Enemy (Jack Reacher, No. 8)', 4.429))]
[('0312873441', ('Running On Instinct', 3.5))]
```

2. Topic-Sensitive PR, when we select the books with average ratings smaller than 4:

```
[('0385504209', ('The Da Vinci Code', 3.841))]
[('0385333927', ('Pagan Babies', 3.625))]
[('0375826688', ('Eragon (Inheritance)', 3.708))]
[('039914563X', ('The Bear and the Dragon', 2.2))]
[('0871138646', ('Old Flames', 3.5))]
[('0373250517', ('Spitting Feathers (Red Dress Ink)', 5.0))]
[('0312873441', ('Running On Instinct', 3.5))]
[('0345464982', ('Bad Girl: A Novel', 3.667))]
[('0618071687', ("Mrs. Hollingsworth's Men", 3.25))]
[('0399148701', ('Red Rabbit', 2.385))]
```

As we can see, introducing the *average ratings* does indeed change the distribution of the random surfers across the nodes! By adding the ratings, readers will have larger probability of being assigned towards higher, or lower, rated books with respect to the baseline.

7 Declaration of Own Work

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.