Homework 2

Francesco Ilario 1469228
Marzio Monticelli 1459333

May 25, 2017

1 Dataset

The used dataset is composed by a graph where each node represents a movie. The Movie-Graph is composed by 1.682 nodes and 983.206 edges. For what concerns the `user_movie_rating.txt`, it contains 943 users, 100.000 ratings and 1.682 different rated movies, so all the movies in the graph has been rated by at least one user.

2 Pagerank Algorithm

In the following we will present the developed functions used in order to calculate the Pagerank for an undirected weighted graph. They are two functions: the main one and the auxiliary one.
The former is the `calculate_page_rank()` function. It is used to trigger the process that will calculate the Pagerank. It needs no parameter to be tuned, but it gives the possibility to declare the graph on which you would calculate the Pagerank (by default it calls a function that will extract the graph from the given `movie_graph.txt` file), the alpha value (by default it is 0.15), the epsilon, i.e. the degree of convergence (by default it is 10^{-6}), and the personalization vector (by default it is None). First of all it calculates an initial Pagerank distributing the probability in an uniform way among all nodes. So it creates a vector where each node has an entry and each node has the same value of Pagerank that is equal to: $\frac{1}{N}$, where N is the total number of nodes in the graph. After that the function will create a vector named `precomputed_weights`, in which for each node it will store the sum of all its edges' weights. This vector will be used as an optimization by the auxiliary method in order to do not recalculate it every time. Finally it will enter in a loop where it will call the auxiliary method that will give back the next-iteration Pagerank, and will end when the distance between two following Pageranks will be less than `epsilon`.

Pagerank Main Function for Undirected Weighted Graph

```
def calculate_page_rank(graph=dr.extract_graph(), alpha=alpha, epsilon=epsilon,
                        personalization=None):
    """
    It computes the pagerank for the given Undirected Weighted Graph, alpha, epsilon,
    and personalization.

    :param networkx.Graph graph: Weighted Undirected graph
    :param float alpha: the pagerank alpha value
    :param float epsilon: the degree of convergence
    :param dict[int, float] personalization: a map <node_id, interest> used to personalize
     the teleport
    :return: the pagerank with a degree of convergence of epsilon
    """
    # creates an initial uniform distribution of probability among all nodes
    previous_page_rank_vector = create_initial_pagerank_vector(graph)

    num_iterations = 0
    # pre-compute weights vector for optimization
    precomputed_weights = {}
    for node_j in graph.nodes(data=False):
        if node_j not in precomputed_weights:
            weight_tot = 0.
            for adj2_node_j in graph.neighbors(node_j):
                data = graph.get_edge_data(node_j, adj2_node_j)
                assert 'weight' in data.keys()
                weight_tot += float(data['weight'])
            precomputed_weights[node_j] = weight_tot

    while True:
        # compute next pagerank from the previous one
        pagerank_vector = single_iteration_page_rank(graph, previous_page_rank_vector,
                                                     alpha=alpha, personalization=personalization,
                                                     precomputed_weights=precomputed_weights)

        num_iterations += 1
        # evaluate the distance between the old and new pagerank vectors
        distance = get_distance(previous_page_rank_vector, pagerank_vector)
        # check convergency
        if distance <= epsilon:
            # print
            # print " Convergence!"
            # print
            break

        previous_page_rank_vector = pagerank_vector

    # sort the pagerank vector in decreasing order of pagerank value
    sorted_pagerank_vector = sorted(pagerank_vector.items(), key=operator.itemgetter(1),
                                    reverse=True)
    sorted_pagerank_dict = {}
    for pr_tuple in sorted_pagerank_vector:
        sorted_pagerank_dict[pr_tuple[0]] = pr_tuple[1]

    return sorted_pagerank_dict
```

The latter method -the auxiliary one- calculates the next Pagerank from the previous one. The product of this operation is a more precise Pagerank estimation based on the previous one. It keeps five parameters:

1. `graph`: it is the graph on which the Pagerank must be calculated;
2. `page_rank_vector`: the previous Pagerank;
3. `alpha`: expresses how to split the probability between the normal behavior and the teleport (by default it is 0.15);
4. `personalization`: a vector that express how to personalize the teleport (by default it is None);
5. `precomputed_weights`: an optimization used in order to do not recalculate every time the sum of all the edges' weights of every node (by default it is None).

The main differences between this algorithm and the given one for the Direct Unweighted Graph are the facts that -obviously- with a Undirected graph we will no more distinguish between *in* and *out* edges, the different way to calculate the "importance" ratio of an edge (Algorithm 2, line 43), and the possibility to personalize the teleport. The Unweighted algorithm can be thought as a generalization of the Weighted where each edge has unitary weight. At each iteration, the new Pagerank is obtained from the previous Pagerank. The new Pagerank value of a node is obtained by considering each outgoing edge and iteratively summing the previous Pagerank value scaled by the *damping factor* $(1 - \alpha)$ and by the importance of the considered edge. More formally, having the previous Pagerank value P_n for the node n, the next Pagerank value P'_n will be:

$$P'_n = \sum_{e_{n,m} \in E_n} weight(e_{n,m}) \times (1 - \alpha) \times importance(e_{n,m})$$

The importance of an edge $e_{n,m}$ that connects two nodes (n, m), is calculated in the Undirected Weighted Graph as follows:

$$importance(e_{n,m}) = \frac{weight(e_{n,m})}{\sum_{e_{m,p} \in E_m} weight(e_{m,p})}$$

The Unweighted Graph can be thought as a particular case of the Weighted Graph, where each edge has weight 1. Hence, the *importance* in the Unweighted Graph case will be the follow:

$$importance(e_{n,m}) = \frac{1}{|E_m|}$$

Indeed in this case $weight(e_{n,m})$ is equal to 1, and $\sum_{e_{m,p} \in E_m} weight(e_{m,p})$ is equal to $|E_m|$. The last important difference is the *personalization* dictionary. It is a dictionary with as key a node id, and as value a float that expresses the interest in teleporting to this node. It is not necessary to normalize this values, they will be normalized by the algorithm. If the personalization dictionary is not defined, the teleport will be calculated uniformly among all nodes.

Pagerank Auxiliary Function for Undirected Weighted Graph

```python
def single_iteration_page_rank(graph, page_rank_vector, alpha=alpha, personalization=None,
                               precomputed_weights=None):
    """
    Calculates a single iteration pagerank from the previous pagerank

    :param networkx.Graph graph:
    :param dict[int, float] page_rank_vector: previous iteration pagerank
    :param float alpha: the alpha value of the pagerank
    :param dict[int, float] | None personalization: a map <node_id, interest> used to
     personalize the teleport
    :param dict[int, float] precomputed_weights: it is an optimization in order to speed up
     the algorithm using the Dynamic Programming technique. Without this optimization the sum
     of the weights of all the edges of each node will be computed at each iteration, while
     with this optimization it would not be needed and the algorithm will be much faster.
    :return: the current iteration pagerank
    :rtype: dict[int, float]
    """
    if precomputed_weights is None:
        precomputed_weights = dict()
    next_page_rank_vector = {}
    sum_rjs = 0.
    for node_j in graph.nodes(data=False):
        r_j = 0.
        for adj_node_j in graph.neighbors(node_j):
            data = graph.get_edge_data(node_j, adj_node_j)
            assert 'weight' in data.keys()
            weight = float(data['weight'])

            # retrieve or calculate the sum of the weights of the edges of the adjacent node
            if adj_node_j in precomputed_weights:
                weight_tot = precomputed_weights[adj_node_j]
            else:
                weight_tot = 0.
                for adj2_node_j in graph.neighbors(adj_node_j):
                    data = graph.get_edge_data(adj_node_j, adj2_node_j)
                    assert 'weight' in data.keys()
                    weight_tot += float(data['weight'])
                precomputed_weights[adj_node_j] = weight_tot

            # increases the pagerank value of the current node with the contribute from the
            # edge in analysis
            r_j += (1. - alpha) * page_rank_vector[adj_node_j] * weight / weight_tot
        next_page_rank_vector[node_j] = r_j
        sum_rjs += r_j
    leaked_pr = 1. - sum_rjs

    # here it divides the leaked_pr
    # (the alpha % of probability that has to be given to the Teleport)
    if personalization:
        assert type(personalization) == dict
        personalization_total_values = float(sum(personalization.values()))

        # distributes the probability with respect to the preferences
        # expressed with the personalization vector
        for node_index, personalization_value in personalization.items():
            assert node_index in next_page_rank_vector
            next_page_rank_vector[
                node_index] += leaked_pr * personalization_value /
                     personalization_total_values

    else:
        # uniform distribution among all nodes
        num_nodes = graph.number_of_nodes()
        for index in next_page_rank_vector.keys():
            next_page_rank_vector[index] += leaked_pr / num_nodes

    # check: the sum of all the PageRanks must be 1.0
    # because it is a probability distribution
    next_page_rank_vector_sum = float(sum(next_page_rank_vector.values()))
    assert is_close(next_page_rank_vector_sum, 1.)

    return next_page_rank_vector
```

3 Part Two

The goal of this part is to implement a method for recommending movies to users contained in the file "user_movie_rating.txt". When started the program parses the inputs and checks for correctness, then creates a personalization vector with the data extracted for the given users.

Pagerank with User-Movie-Ratings: Personalization

```
# extract personalization from user-rating
personalization = {}
ratings_sum = float(sum(user_rated_movies.values()))

# normalization
for movie_id, rating in user_rated_movies.items():
    personalization[movie_id] = rating / ratings_sum
```

This personalization vector is now used to calculate the Pagerank.

Pagerank with User-Movie-Ratings: Calculation

```
user_page_rank = po.calculate_page_rank(graph=movie_graph, personalization=personalization)
```

As last thing, from the resultant Pagerank are filtered out all the entry for the user's yet-rated films and the Pagerank is normalized to 1.0.

Pagerank with User-Movie-Ratings: Filter/Normalization

```
# Filter user_page_ranks:
for movie_id in user_rated_movies.keys():
    user_page_rank.pop(movie_id)

# renormalize pagerank to 1.0
pagerank_sum = float(sum(user_page_rank.values()))
for movie_id, pagerank in user_page_rank.items():
    user_page_rank[movie_id] = pagerank / pagerank_sum
```

4 Part Three

In this part the user can express its interest in the available categories (5 for this dataset) through a vector of the form

$$\textit{WeightCategory1\_WeightCategory2\_...\_WeightCategory5.}$$

The program have to build the personalization vector from the user input and return the topic-specific PageRank. Another important thing is that the calculation of the Pagerank must be done in two phases:

1. `Offline`: must generate independent data that will be used later;
2. `Online`: uses the offline generated independent data to obtain the Pagerank, without really calculating it.

In order to do this, we can rely on what said in the Section 21.2 of the Book *Introduction to Information Retrieval*:
"It is not necessary to compute a PageRank vector for every distinct combination of user interests over topics; the personalized PageRank vector for any user can be expressed as a linear combination of the underlying topic-specific PageRanks. For instance, the personalized PageRank vector for the user whose interests are 60% sports and 40% politics can be computed as:

$$0.6\pi_s + 0.4\pi_p$$

where π_s and π_p are the topic-specific PageRank vectors for sports and politics, respectively.".
Hence, we can pre-compute the PageRank of each category in the offline part, and linearly combine them at request time (online).

4.1 Offline

The main part of the Offline module is the function `create_datasets()`:

Category-Pagerank Online: Datasets Creation

```
1  def create_datasets():
2      categories = dr.extract_categories()
3
4      for cat_num, category in enumerate(categories):
5          cat_file_path = dataset_dir + category_file_prefix + str(cat_num)
6          if os.path.isfile(cat_file_path):
7              continue
8
9          cat_file = open(cat_file_path, 'w')
10
11         cat_personalization = {0: 0., 1: 0., 2: 0., 3: 0., 4: 0., cat_num: 1.}
12         category_pagerank =
               po.calculate_page_rank_categories(category_interest=cat_personalization)
13
14         for movie_id, movie_pagerank in category_pagerank.items():
15             cat_file.write(str(movie_id) + ": " + str(movie_pagerank) + "\n")
16
17         cat_file.close()
```

This function will store inside an ad-hoc file the PageRank for each category. The auxiliary function `calculate_page_rank_categories()` simply creates the personalization vector in the form <node_id, interest> from the one in the form <category_id, interest> and then calls the PageRank function.

Category-Pagerank Function

```
def calculate_page_rank_categories(graph=dr.extract_graph(),
      categories=dr.extract_categories(), category_interest=None, alpha=alpha):
    """
    :param networkx.Graph graph: movie graph
    :param dict[int, list[int]] categories: relation category_id <-> movie_id
    :param dict[int, float] category_interest: float values for interests in category
    :return: a PageRank vector: one Real value for each node (movie_id) of the graph
    :rtype: dict[int, float]
    """
    personalization = None
    if category_interest:
        personalization = from_preference_to_weights(categories, category_interest)

    return calculate_page_rank(graph=graph, personalization=personalization, alpha=alpha)
```

4.2 Online

For what concerns the Online part, first of all it checks if the category-PageRank files have yet been calculated, otherwise it ask to the offline module to generate them.

Category-Pagerank Online: Check

```
if not offline.are_category_pagerank_generated():
    offline.create_datasets()
```

Then it normalizes the input vector.

Category-Pagerank Online: Interest Normalization

```
# normalize categories_interest
interest_sum = float(sum(user_preference_vector_float.values()))
for cat_id in user_preference_vector_float.keys():
    user_preference_vector_float[cat_id] /= interest_sum
```

Now we are sure to have the files we need in order to calculate the result. So we linearly combine the previously calculated PageRanks scaling each of them for the normalized interest in the corresponding category.

Category-Pagerank Online: Result Composition

```
# Calculating composite pagerank
num_pagerank_entries = len(offline.get_category_pagerank(0).keys())
composite_pagerank = dict((x, 0.) for x in xrange(1, num_pagerank_entries + 1))

for cat_id, interest in user_preference_vector_float.items():
    if interest > 0.:
        cat_pagerank = offline.get_category_pagerank(cat_id)
        for comp_cat_id, comp_interest in composite_pagerank.items():
            composite_pagerank[comp_cat_id] += cat_pagerank[comp_cat_id] * interest
```