Massive Data Analytics

Home Work 4

Ilia Hashemi Rad

99102456

Many Explanations about codes used or algorithms for recommendation systems were given in the report of previous HomeWork. So, I avoid to give them again here; But I attach the previous report with my files again for this Homework

Random Walk Algorithms:

General Explanations

Explanations:

Random walk algorithms are a type of stochastic process that involves moving from one state to another in a probabilistic way. They can be used to model various phenomena such as diffusion, gambling, and network exploration. In the context of recommendation systems, random walk algorithms can help to find items that are relevant to a user's preferences or interests, based on the relationships between users and items, or between items and other items.

One way to use random walk algorithms for recommendation systems is to construct a graph that represents the user-item interactions, such as ratings, purchases, clicks, etc. Each node in the graph can be either a user or an item, and each edge can have a weight that reflects the strength of the interaction. For example, a higher rating or a more frequent purchase can indicate a stronger preference. Then, a random walk can be performed on the graph, starting from a given user node, and following the edges with some probability that depends on the edge weights. The random walk can visit multiple nodes, and keep track of the visited items and their frequencies. The items that are visited more often by the random walk can be considered as more relevant or similar to the user's preferences, and thus recommended to the user.

Another way to use random walk algorithms for recommendation systems is to construct a graph that represents the item-item similarities, based on some features or attributes of the items, such as genre, category, keywords, etc. Each node in the graph can be an item, and each edge can have a weight that reflects the similarity between the items. For example, a higher similarity score or a more common feature can indicate a stronger connection. Then, a random walk can be performed on the graph, starting from a given item node, and following the edges with some probability that depends on the edge weights. The random walk can visit multiple nodes, and keep track of the visited items and their frequencies. The items that are visited more often by the random walk can be considered as more related or complementary to the given item, and thus recommended to the user.

Random walk algorithms have some advantages and disadvantages for recommendation systems. Some of the advantages are:

- They can capture the complex and nonlinear relationships between users and items, or between items and other items, that may not be easily modeled by other methods.
- They can handle sparse and dynamic data, as they do not require a complete or fixed matrix of user-item interactions or item-item similarities.
- They can provide diverse and serendipitous recommendations, as they can explore different parts of the graph and discover new or unexpected items.

Some of the disadvantages are:

- They can be computationally expensive, as they require multiple iterations and random choices to generate recommendations.
- They can be sensitive to the choice of parameters, such as the length of the random walk, the probability distribution of the edge weights, and the stopping criterion.

- They can be influenced by noise or outliers, as they may follow some random or irrelevant paths that lead to poor recommendations.

Random Walk with Restart algorithm (RWR)

Explanations:

Random walk with restart (RWR) is a variation of random walk that allows the walker to return to the starting node with some probability at each step. This way, the walker can explore the local neighborhood of the starting node more thoroughly, and avoid wandering too far away from it. RWR can be used for ranking and link prediction on graphs, based on the idea that nodes that are frequently visited by the walker are more relevant or similar to the starting node.

One way to use RWR for ranking is to compute the stationary distribution of the walker, which represents the probability of finding the walker at each node after a large number of steps. The stationary distribution can be obtained by solving a linear system of equations, or by using an iterative method such as power iteration. The nodes with higher probabilities in the stationary distribution can be ranked higher as recommendations for the starting node.

RWR has some advantages and disadvantages for ranking and link prediction. Some of the advantages are:

- It can capture the global structure of the graph, as well as the local similarity between nodes.
- It can handle weighted and directed graphs, by adjusting the edge weights and the restart probability accordingly.
- It can incorporate prior knowledge or preferences, by using different restart probabilities for different nodes.

Some of the disadvantages are:

- It can be computationally expensive, as it requires solving a linear system or performing multiple matrix-vector multiplications.
- It can be sensitive to the choice of the restart probability, which affects the trade-off between exploration and exploitation.
- It can be influenced by the degree distribution of the graph, as high-degree nodes tend to have higher probabilities in the stationary distribution.

My General Approach:

Explanations:

I chose RWR algorithm because of handling False Positives which in a twitter recommendation system is more important than False Negatives in my mind. Because, when you have much false negative in twitter (but not news), the user is probable to don't see some of his attractive tweets which is not such a bad thing cause he is still seeing some of his attractive ones; But, when we have much false positives, it means the user may be recommended to see some tweets which are not attractive to him at all, or may even be disgusting to him, resulting or encouraging him to use the Twitter less than before. This is too bad. So, I decided to pay attention to FP more than FN by using RWR.

This algorithm, because of having a restart probability in each step, which was explained previously, reduces the false positives and makes the system have a more reliable and relevant suggestions.

The goal of my random walk algorithm is to find the top-k similar users to a given user based on his interactions on other's tweets. The algorithm consists of four main steps:

1. First, the algorithm creates a graph from users' interactions with each other, named `user_rating` RDD (which is also explained further), where each node is a user and each edge is weighted by the rating. The rating can be 1, 2, or 3 depending on the tweet type (replied, retweeted, or quoted). The graph represents the network of users and their preferences.

2. Second, the algorithm performs a random walk with restart on the graph, starting from the target user. A random walk is a process of randomly choosing a neighbor of the current node based on the edge weights and moving to that node. A random walk with restart is a variation of random walk that restarts the process from the target user with some probability at each step. This ensures that the random walk does not stray too far from the target user and explores the local neighborhood of the graph.

3. Third, the algorithm counts the number of visits of each node after performing the random walk with restart for a given number of steps. The visit count of a node reflects the probability of reaching that node from the target user by following the random paths on the graph. The higher the visit count, the more similar the node is to the target user.

4. Fourth, the algorithm sorts the nodes by their visit counts in descending order and returns the first k nodes, excluding the target user. These are the top-k similar users to the target user based on the random walk algorithm.

Section 1: Preprocessing and Exploring data

```python
tweets_rdd = sc.textFile("twitter_data_v2.jsonl")

import json

def create_tree(tweet_data):
    tree = {}

    # Extracting top-level fields
    tree["id"] = tweet_data["id"]
    tree["user"] = {
        "id": tweet_data["user"]["id"],
        "screen_name": tweet_data["user"]["screen_name"],
        "name": tweet_data["user"]["name"],
        "description": tweet_data["user"]["description"]
    }
    tree["tweet_type"] = tweet_data["tweet_type"]
    tree["in_reply_to_status_id_str"] = tweet_data["in_reply_to_status_id_str"]
    tree["in_reply_to_user_id_str"] = tweet_data["in_reply_to_user_id_str"]

    # Quoted status
    quoted_status = tweet_data.get("quoted_status")
    if quoted_status:
        tree["quoted_status"] = {
            "id": quoted_status["id"],
        }

    # Retweeted status
    retweeted_status = tweet_data.get("retweeted_status")
    if retweeted_status:
        tree["retweeted_status"] = {
            "id": retweeted_status["id"],
            "created_at": retweeted_status["created_at"],
            "user": {
                "id": retweeted_status["user"]["id"],
                "screen_name": retweeted_status["user"]["screen_name"],
                "name": retweeted_status["user"]["name"],
                "description": retweeted_status["user"]["description"]
            }
        }

    return tree

# Example for retweeted data
sample_data = json.loads(tweets_rdd.take(10)[0])
tweet_tree = create_tree(sample_data)
print(json.dumps(tweet_tree, indent=2, ensure_ascii=False))
```

Explanations:

Here, in this code, I Preprocessed the data to filter the relevant parts of data which was indicated in the explanations of assignment, and showed that in a hierarchical or tree like shape. That was done for all the 4 types of tweet to get more familiar with them and know how to determine ratings of each user for an specific tweet. Here, is a sample output of that, for each type:

Retweeted Tweets


```
{
  "id": "1736536753815834746",
  "user": {
    "id": "2808957837",
    "screen_name": "maryaa_maryam",
    "name": "maryam jafari",
    "description": ""
  },
  "tweet_type": "retweeted",
  "in_reply_to_status_id_str": null,
  "in_reply_to_user_id_str": null,
  "retweeted_status": {
    "id": "1736379038376120479",
    "created_at": 1702819997,
    "user": {
      "id": "2960042892",
      "screen_name": "Ivar_lathbrug2",
      "name": "آیوار ۲",
      "description": "باورمند به نظام پادشاهی مشروطه با @PahlaviReza #جاویدشاه @PhoenixprjIran https://t.co/kjMxf53cIQ"
    }
  }
}
```

Replied Tweets


```
{
  "id": "1736536759964512322",
  "user": {
    "id": "1406742149048287242",
    "screen_name": "sun_saeid",
    "name": "(تلخ) حق نامه",
    "description": "از زندگی بگیم ولی واقعیتشو"
  },
  "tweet_type": "replied",
  "in_reply_to_status_id_str": "1736356968162378084",
  "in_reply_to_user_id_str": "1643210821068046338"
}
```


Generated Tweets


```
{
  "id": "1736536756835512564",
  "user": {
    "id": "22495731",
    "screen_name": "AlArabiya_Fa",
    "name": "العربیه فارسی",
    "description": "«راه‌اندازی شد» 2008 در سال «العربیه» به عنوان بخشی از شبکه «العربیه فارسی». \nایمیل تحریریه العربیه فارسی:\neditor.farsi@alarabiya.net\nhttp
  },
  "tweet_type": "generated",
  "in_reply_to_status_id_str": null,
  "in_reply_to_user_id_str": null
}
```


Quoted Tweets


```
{
  "id": "1736536759943450888",
  "user": {
    "id": "1655373597001523200",
    "screen_name": "ScarFace2582",
    "name": "SCARFACE ♔",
    "description": "♔\n#جاویدشاه #پاینده_ایران #ایران_معبد_ماست\n♔"
  },
  "tweet_type": "quoted",
  "in_reply_to_status_id_str": null,
  "in_reply_to_user_id_str": null,
  "quoted_status": {
    "id": "1736113075911077948"
  },
  "retweeted_status": {
    "id": "1736536140063109215",
    "created_at": 1702857453,
    "user": {
      "id": "893757772025798656",
      "screen_name": "YaarDabestaani",
      "name": "Yaar - یار دبستانی 🏳️‍🌈",
      "description": "Human rights activist, 🌈LGBTQ🏳️‍🌈, animal lover🐕🐈‍⬛✊🏾, Atheist, 🤦🏻‍♂️=👦🏻💼=🤦🏻‍♀️=🌑💼 #MahsaAmini,  My inner child: @bri30s"
    }
  }
}
```

Section 2: Determining a Dictionary of Reference Tweet of Quoted Tweets

```
import json

tweets_rdd = sc.textFile("twitter_data_v2.jsonl")

# Parse JSON strings to Python dictionaries
tweets_json = tweets_rdd.map(json.loads)
del tweets_rdd

# Create an RDD of quoted tweet ids
quoted_tweets_ids = tweets_json.filter(lambda x: x['tweet_type'] == "quoted").map(lambda x: (x.get('quoted_status').get('id'), None
# Join the original RDD with the quoted tweet ids RDD
quoted_rdd = tweets_json.map(lambda x: (x['id'], x.get('user').get('id'))).join(quoted_tweets_ids)

# Remove the None values from the result
quoted_list = quoted_rdd.map(lambda x: (x[0], x[1][0])).collect()

# Create a dictionary of tweets which are the reference of quoted tweets as (tweet ID, user ID)
quoted_dict = dict(quoted_list)

print(dict(quoted_list[:20]))
```

Explanations:

In my algorithm, as I explained earlier, I want to create a graph of users based on their interaction. So, I should have all the interactions of the users. Here in this code, I prepared some data to do so.

If you look at the quoted tweets as shown in previous section, you will see the `quoted_status` field has just the ID of the tweet quoted from and not its user. But here, we want users' relationship and not items. So, I at first collected all IDs of tweets quoted from (reference tweets), and then, sought them in all tweets to find their users and create a dictionary of their tweet IDs and user IDs.

The result of this part is as (tweet ID, user ID) of the reference tweet of a quoted tweet.

Here is a brief explanation of what I did in the code: (The loading part are explained further and also in HW3)

- **Create RDD of Quoted Tweet IDs:**
  - I filtered the tweets to include only those with the `tweet_type` equal to "quoted."
  - Mapped each tweet to a tuple containing the quoted tweet ID and `None`.
- **Join Original RDD with Quoted Tweet IDs RDD:**
  - I mapped each tweet to a tuple containing the tweet ID and user ID.
  - I joined this RDD with the RDD of quoted tweet IDs using `join` to find the reference tweets of quoted ones.
- **Remove `None` Values:**
  - Removed `None` values from the joined result by mapping it to a tuple containing the tweet ID and user ID.
- **Create Dictionary of Quoted Tweets:**
  - Created a dictionary (`quoted_dict`) from the collected list of tuples, where each entry represents a quoted tweet with the tweet ID as the key and the user ID as the value.
  - Finally, Displayed the first 20 entries of the dictionary.

Here, is the output of the code: (Some samples of the dictionary)

{'1735116408798027896': '1576297375970525184',
'1734416486838493585': '1017869466103877634',
'1735127269465862373': '568002438',
'1735519944362586556': '1128549743855112192',
'1736630273771491555': '1062336497737699328',
'1734085506948894902': '1577551087887851520'}

Section 3: Determining Users' Degree of Relation

```python
import json

tweets_rdd = sc.textFile("twitter_data_v2.jsonl")

# Parse JSON strings to Python dictionaries
tweets_json = tweets_rdd.map(json.loads)
del tweets_rdd

# Filter out generated tweets
relevant_tweets = tweets_json.filter(lambda x: x['tweet_type'] in ["retweeted", "replied", "quoted"])

# Extract users' interaction rate with each other output
def extract_tweet_rates(tweet):
    user_id = tweet.get("user", {}).get("id", None)
    tweet_type = tweet.get("tweet_type", None)
    if tweet_type == 'replied':
        user_ref = tweet.get("in_reply_to_user_id_str", None) # reference user
        return (user_id, user_ref, 1)
    elif tweet_type == 'retweeted':
        user_ref = tweet.get("retweeted_status", {}).get("user", None).get("id", None) # reference user
        return (user_id, user_ref, 2)
    else:
        quoted_tweet_id = tweet.get("quoted_status", {}).get("id", None) # ID of tweet quoted from
        try:
            user_ref = quoted_dict[quoted_tweet_id] # user of tweet quoted from
            return (user_id, user_ref, 3)
        except:
            return None # return none if the reference tweet not found in the data (so we can't find the reference user)

user_rating = relevant_tweets.map(extract_tweet_rates).filter(lambda x: x!= None)

user_rating.takeSample(False, 8, 42)
```

Explanations:

Here is a brief explanation of what I did in the code: (The inference behind the ratings has been discussed in the previous home work)

- **Parse JSON Tweets:**
  - I used the `sc.textFile` function to read a JSON lines file (`twitter_data_v2.jsonl`) into an RDD (`tweets_rdd`).
  - Used `map` to parse each JSON string into Python dictionaries (`tweets_json`).
  - Then, released the memory occupied by the original RDD (`tweets_rdd`) using `del`.
- **Filter Out Generated Tweets:**
  - I filtered out tweets based on their types, keeping only "retweeted," "replied," and "quoted" tweets and filtering "Generated" ones, because we can't extract any rating from them.
- **Extract Users' Interaction Rates:**
  - Defined a function (`extract_tweet_rates`) to extract user interaction rates based on different tweet types.
  - For "replied" tweets, extracted the user ID and the user being replied to (`user_ref`).
  - For "retweeted" tweets, extracted the user ID and the original tweet's user ID (`user_ref`).
  - For "quoted" tweets, extracted the user ID, the ID of the quoted tweet, and the user ID of the original tweet being quoted (`user_ref`). Used the `quoted_dict` to find the user ID of the reference of quoted tweet.

- Filtered out `None` values to exclude cases where the reference tweet or user cannot be found.

- **Display Sample User Ratings:**
  - Each entry represents a tuple with the user ID, reference user ID, and interaction type (1 for replied, 2 for retweeted, 3 for quoted).

Now, we have successfully captured all relationships between users and the degree of that. This can easily be processed into the desired graph for random walk algorithms.

Here, is some samples of `user_rating` which expectedly, has the form of (user ID 1, user ID 2, degree of relation):

```
[('1696062547533684736', '1351371042', 1),
 ('1012004974690275329', '3189754680', 2),
 ('2583526102', '1219846015345483779', 1),
 ('1561681142344044545', '1675560941885222913', 1),
 ('1655986795224367127', '1184763184948957185', 1),
 ('1614593362123710466', '1155366665019908096', 2),
 ('1327700517796114434', '820372916722434048', 2),
 ('1576699553948090368', '1638951842271862791', 1)]
```

Section 4: Finding Similar Users by RWR algorithm

Determining the desired Graph

```
# Create a graph from the user_rating RDD
graph = user_rating.flatMap(lambda x: [(x[0], (x[1], x[2])), (x[1], (x[0], x[2]))]) \
                   .groupByKey() \
                   .mapValues(list) \
                   .collectAsMap()

print({v:graph[v] for v in [list(graph.keys())[k] for k in range(1)]})
```

Explanations:

Here, using the `user_rating` RDD, I made the desired graph for RWR algorithm. The nodes of the graph indicate users, and the edges indicate the degree of relation between the, as explained earlier. Now, Here is the explanation of how I made the graph in code:

- **Create a Graph from User Ratings:**
  - I used the `user_rating` RDD to create a graph representing user interactions.
  - Used `flatMap` to transform each entry in `user_rating` into two key-value pairs: (`user_id`, (`user_ref`, `interaction_type`)) and (`user_ref`, (`user_id`, `interaction_type`)).
  - Used `groupByKey` to group the values by key (user ID or reference user ID).
  - Used `mapValues` to convert the grouped values into lists.
  - Used `collectAsMap` to collect the result as a dictionary (`graph`) where keys are user IDs, and values are lists of tuples representing interactions.
  - Finally, printed a sample of the graph for one key (user ID) for demonstration purposes.

Here is the output: (one sample of the graph)

{'933731737686994946':

[('1287439194663653376', 1), ('45284178', 1), ('1457064158025879554', 1), ('1458463785300148229', 1), ('1596756634818207746', 3), ('495082191', 1), ('495082191', 1), ('430566432', 1), ('1038187447283343361', 1), ('1275069380569657344', 1), ('1451085166970118146', 1), ('1314403966030090242', 1), ('1222946740929617920', 1), ('1550915415110549505', 1), ('1038187447283343361', 1), ('1451085166970118146', 1), ('1038187447283343361', 1), ('1595111714143572000', 1), ('1038187447283343361', 1), ('1038187447283343361', 1), ('1355183936641576965', 1), ('880414261478125569', 1), ('1046105731936776194', 1), ('1296386200497000454', 1), ('1177099107539791872', 1), ('1671085861532753924', 1), ('1671085861532753924', 1), ('1328753786748362759', 1), ('1595111714143572000', 1), ('1296386200497000454', 1), ('1296386200497000454', 1), ('1296386200497000454', 1), ('1444240663143399424', 1), ('1595111714143572000', 1), ('1005696819865808897', 3), ('1595111714143572000', 1), ('1086167707962302464', 3), ('495082191', 1), ('933731737686994946', 1), ('933731737686994946', 1), ('1596756634818207746', 1), ('495082191', 1), ('1444240663143399424', 1), ('1323078091', 1), ('1046105731936776194', 1), ('1457064158025879554', 1), ('1596756634818207746', 1), ('1457064158025879554', 1), ('1458596505355444228', 1), ('1457064158025879554', 1), ('880414261478125569', 1), ('933731737686994946', 3), ('933731737686994946', 3), ('1046105731936776194', 1), ('1296386200497000454', 1), ('1038187447283343361', 1), ('1009086806623744000', 1), ('495082191', 1), ('495082191', 1), ('1444240663143399424', 1), ('1046105731936776194', 1), ('933731737686994946', 1), ('933731737686994946', 1), ('1009086806623744000', 1), ('933731737686994946', 1), ('933731737686994946', 1), ('495082191', 1), ('933731737686994946', 1), ('933731737686994946', 1), ('1046105731936776194', 1), ('1046105731936776194', 1), ('1038187447283343361', 1), ('495082191', 1), ('1355989861958823942', 1), ('1355989861958823942', 1), ('1038187447283343361', 1), ('1038187447283343361', 1), ('1038187447283343361', 1), ('933731737686994946', 1), ('933731737686994946', 1), ('1409844849491255299', 1), ('933731737686994946', 1), ('933731737686994946', 1), ('1596756634818207746', 1), ('1409844849491255299', 1), ('1596756634818207746', 1), ('1596756634818207746', 1), ('1596756634818207746', 1), ('1463552247665680392', 1), ('1046105731936776194', 1), ('1409844849491255299', 1), ('2743967999', 1), ('1409844849491255299', 1), ('2462027260', 1), ('2994218416', 1), ('2257605577', 1), ('1596756634818207746', 1), ('1296386200497000454', 1), ('1596756634818207746', 1), ('1596756634818207746', 1), ('1324030313505443841', 2), ('1444240663143399424', 1), ('2462027260', 1), ('1444240663143399424', 1), ('818847733767409664', 1), ('1409949055166341123', 1), ('1463552247665680392', 1), ('1463552247665680392', 1), ('1296386200497000454', 1), ('1463552247665680392', 1), ('933731737686994946', 3), ('933731737686994946', 3), ('1046105731936776194', 1), ('1210363371473780738', 1), ('1038187447283343361', 1), ('1541423555593834502', 2)]

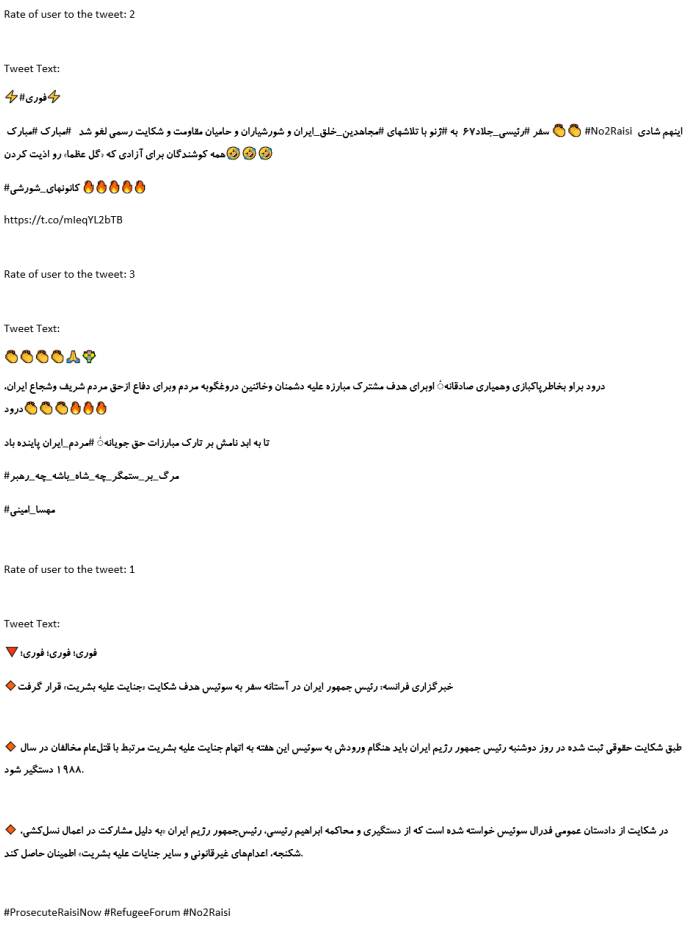
}

Choosing One target User And Showing his Favorite Tweets

Explanations:

In this Home Work, I chose the user which was chosen in the previous home work to satisfy the resluts asked in the home work and be able to compare the results of this algorithm with the Collaborative filtering one. I had determined some samples of the user's preferences in HW3. So, I just show them here to be reminded.

Here, is one shot of some samples of the target user's favorite tweets: (complete output can be seen in the HW3 Jupyter Notebook)



Rate of user to the tweet: 2

Tweet Text:

⚡#فوری⚡

اینهم شادی #No2Raisi 👏👏 سفر #رئیسی_جلاد۶۷ به #ژنو با تلاشهای #مجاهدین_خلق_ایران و شورشیاران و حامیان مقاومت و شکایت رسمی لغو شد #مبارک #مبارک
همه کوشندگان برای آزادی که «گل عظما» رو اذیت کردن🤣🤣🤣

#کانونهای_شورشی🔥🔥🔥🔥🔥

https://t.co/mleqYL2bTB

Rate of user to the tweet: 3

Tweet Text:

👏👏👏👏🙏🌹

درود براو بخاطرپاکبازی وهمیاری صادقانه♘ اوبرای هدف مشترک مبارزه علیه دشمنان وخائنین دروغگوبه مردم وبرای دفاع ازحق مردم شریف وشجاع ایران،
درود👏👏👏🔥🔥🔥

تا به ابد نامش بر تارک مبارزات حق جویانه♘ #مردم_ایران پاینده باد

#مرگ_بر_ستمگر_چه_شاه_باشه_چه_رهبر

#مهسا_امینی

Rate of user to the tweet: 1

Tweet Text:

🔻فوری؛ فوری؛ فوری؛

🔸خبرگزاری فرانسه: رئیس جمهور ایران در آستانه سفر به سوئیس هدف شکایت جنایت علیه بشریت، قرار گرفت

🔸 طبق شکایت حقوقی ثبت شده در روز دوشنبه رئیس جمهور رژیم ایران باید هنگام ورودش به سوئیس این هفته به اتهام جنایت علیه بشریت مرتبط با قتل‌عام مخالفان در سال ۱۹۸۸ دستگیر شود.

🔸 در شکایت از دادستان عمومی فدرال سوئیس خواسته شده است که از دستگیری و محاکمه ابراهیم رئیسی، رئیس‌جمهور رژیم ایران «به دلیل مشارکت در اعمال نسل‌کشی، شکنجه، اعدام‌های غیرقانونی و سایر جنایات علیه بشریت» اطمینان حاصل کند.

#ProsecuteRaisiNow #RefugeeForum #No2Raisi

Explanations:

As you can see, the user mostly likes the political tweets and has a specific politic side which can be seen from his attractive tweets. Keep that in mind, to consist the recommended tweets with these and see the similarity.

Applying RWR algorithm on the Graph

```
import random

# Define a function to perform one step of random walk
def random_walk_step(node, graph):
    # Get the neighbors and weights of the current node
    neighbors, weights = zip(*graph[node])
    # Randomly choose a neighbor based on the weights
    next_node = random.choices(neighbors, weights=weights)[0]
    # Return the next node
    return next_node

# Define a function to perform multiple steps of random walk with restart
def random_walk_with_restart(node, graph, steps, restart_prob):
    # Initialize a dictionary to store the visit counts of the nodes
    visit_count = {n: 0 for n in graph}
    # Set the current node to the target node
    current_node = node
    # Loop for the given number of steps
    for i in range(steps):
        # Update the visit count of the current node
        visit_count[current_node] += 1
```

```
        # Generate a random number between 0 and 1
        r = random.random()
        # If the random number is less than the restart probability, restart the random walk from the target node
        if r < restart_prob:
            current_node = node
        # Otherwise, perform one step of random walk
        else:
            current_node = random_walk_step(current_node, graph)
    # Return the visit count dictionary
    return visit_count

# Define a function to find the top-k similar users to a given user
def find_similar_users(node, graph, steps, restart_prob, k):
    # Perform random walk with restart from the given node
    visit_count = random_walk_with_restart(node, graph, steps, restart_prob)
    # Sort the nodes by their visit counts in descending order
    sorted_nodes = sorted(visit_count, key=visit_count.get, reverse=True)
    # Return the first k nodes, excluding the target node
    return sorted_nodes[1:k+1]

# Set the target user id
target_user_id = '165605880'
# Set the number of steps, restart probability, and k
steps = 10000
restart_prob = 0.05
k = 50
# Find the top-k similar users to the target user
similar_users = find_similar_users(target_user_id, graph, steps, restart_prob, k)
# Print the result
print(f"The top-{k} similar users to user {target_user_id} are: {similar_users}")
```

Explanations:

- **Random Walk with Restart for Finding Similar Users:**
  - I defined a function `random_walk_step` to perform one step of random walk based on the given node and graph. The next node is randomly chosen among neighbors according to weights of edges which the node is connected to. This makes the users with more interaction degree (for example the users which the tweet of target user is quoted from them), more likely to be chosen in each step.
  - Defined a function `random_walk_with_restart` to perform multiple steps of random walk with a restart probability which was explained earlier. It initializes a visit count dictionary, performs random walks, and updates the visit counts.
  - Defined a function `find_similar_users` to find the top-k similar users to a given node. It uses `random_walk_with_restart` and returns the sorted nodes based on visit counts, excluding the target node.
  - Set the target user ID, number of steps, restart probability, and k. I set the number of steps, based on some studies I had from different sources for RW recommendation systems, set number of top similar users, based on previous HW, and set the restart probability, based on weights of edges connected to the start node. Because they are a lot, the mean probability of a node being selected as the next node, will become about 0.01 according to normalization of weights. So, the restart probability, must be set something like 0.05, to indifferently, we don't go 5 nodes further and have low FP and FN simultaneously.
  - Found the top-k similar users to the target user and print the result.

Here, is the results of 50 found similar users:

The top-50 similar users to user 165605880 are:

['931938860963061760', '2221999644', '852104627566899200', '1640268020374294529', '3390890626', '1555561052', '1624682333394595841', '1430212798567563264', '1193634420000993280', '1096520546987184130', '947844496225644545', '1300119897737961474', '806048003672985601', '2184707526', '82311780', '1269647773307199488', '69058391', '1094620930557517832', '1717719501784170496', '1702354885063192579', '870292251389382657', '1706920934383276032', '1702217750054162432', '1678765614700318721', '845381446516822016', '3554230816', '988405852037156864', '865329625810980865', '1551110542412087297', '971328533275512832', '1617465622111129600', '848911278324273153', '2723935129', '1656228998228189184', '1276187974070403073', '841396621698985984', '2318855006', '1379875551356260361', '1666823496054501376', '2919269142', '1691082595549806592', '1643280415401099264', '868736088755453953', '1607679663060525056', '1695321628912025601', '2205932795', '1162722851755106304', '786738726466707460', '968977423571259392', '1708263425346830336']

Are the recommended users for both CF and RWR algorithms the same?

Explanations:

The answer is **No** in Here and **May not** in general. The top 50 similar users found by the Random Walk with Restart (RWR) algorithm and the top 50 similar users found by the collaborative filtering algorithm may differ for a given user, depending on the data and the parameters of the algorithms. Because the first one is stochastic and the second is deterministic.

The RWR algorithm is based on simulating random paths on a graph of users and items, and measuring the probability of reaching a user from another user. The collaborative filtering algorithm is based on finding users who have rated items similarly, and predicting the ratings of unknown items for a user. These two algorithms have different assumptions and objectives, and may capture different aspects of user similarity.

For example, the RWR algorithm may find users who have interacted with the same items, regardless of their ratings, while the collaborative filtering algorithm may find users who have rated the same items similarly, regardless of their other preferences. The RWR algorithm may also find users who are indirectly connected through multiple hops on the graph, while the collaborative filtering algorithm may only find users who have rated some common items. The RWR algorithm may also be influenced by the restart probability, which controls how often the random walk returns to the target user, while the collaborative filtering algorithm may be influenced by the similarity measure, which determines how to compare the ratings of users.

Therefore, the top 50 similar users found by the RWR algorithm and the top 50 similar users found by the collaborative filtering algorithm may not be the same for a given user. However, they may also have some overlap (and in our case they have), as both algorithms try to find users who have similar preferences or behaviors. The degree of overlap may depend on the characteristics of the data and the parameters of the algorithms.

Section 5: Evaluation

Recommend Some Tweets based on Similar Users

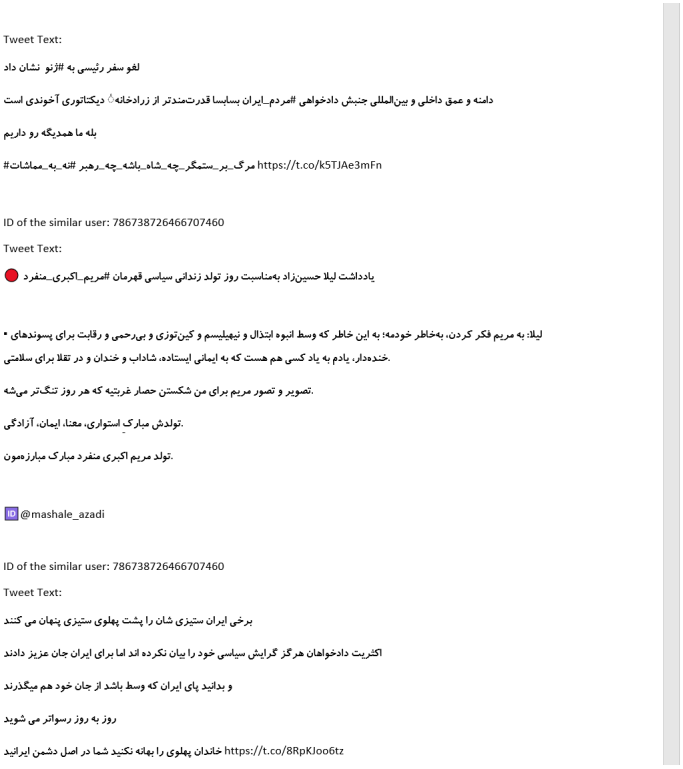
```
top_recommended_tweets = tweets_json.filter(lambda x: x.get('user').get('id') in similar_users).map(lambda x: (x.get('user').get('id'), x.get('text')))

for tweet in top_recommended_tweets[:50]:
    print(f'ID of the similar user: {tweet[0]}\n')
    print('Tweet Text:')
    print(tweet[1])
    print('\n\n')
```

Explanations:

Here is a brief explanation of what I did in the code:

- **Top Recommended Tweets from Similar Users:**
  - I filtered tweets from `tweets_json` RDD based on whether the user ID is in the list of similar users to catch the tweets of similar users.
  - Mapped the filtered tweets to a tuple containing the user ID and tweet text.
  - Collected the top recommended tweets into the `top_recommended_tweets` variable.
  - Printed the user ID and tweet text for the first 50 recommended tweets.

Here, is one shot of top recommended tweets' text with their respective user ID: (a more complete output can be seen in the Jupyter Notebook)



Tweet Text:

لغو سفر رئیسی به #ژنو نشان داد

دامنه و عمق داخلی و بین‌المللی جنبش دادخواهی #مردم_ایران بسابسا قدرتمندتر از زرادخانه♢ دیکتاتوری آخوندی است

بله ما همدیگه رو داریم

#مرگ_بر_ستمگر_چه_شاه_باشه_چه_رهبر #نه_به_مماشات https://t.co/k5TJAe3mFn

ID of the similar user: 786738726466707460

Tweet Text:

یادداشت لیلا حسین‌زاد به‌مناسبت روز تولد زندانی سیاسی قهرمان #مریم_اکبری_منفرد

لیلا: به مریم فکر کردن، به‌خاطر خودمه؛ به این خاطر که وسط انبوه ابتذال و نیهیلیسم و کین‌توزی و بی‌رحمی و رقابت برای پسوندهای •
خنده‌دار، یادم به یاد کسی هم هست که به ایمانی ایستاده، شاداب و خندان و در تقلا برای سلامتی.

تصویر و تصور مریم برای من شکستن حصار غربتیه که هر روز تنگ‌تر می‌شه.

تولدش مبارک استواری، معنا، ایمان، آزادگی.

تولد مریم اکبری منفرد مبارک مبارزمون.

@mashale_azadi

ID of the similar user: 786738726466707460

Tweet Text:

برخی ایران ستیزی شان را پشت پهلوی ستیزی پنهان می کنند

اکثریت دادخواهان هرگز گرایش سیاسی خود را بیان نکرده اند اما برای ایران جان عزیز دادند

و بدانید پای ایران که وسط باشد از جان خود هم میگذرند

روز به روز رسواتر می شوید

خاندان پهلوی را بهانه نکنید شما در اصل دشمن ایرانید https://t.co/8RpKJoo6tz

Tweet Text:

باید به آخوندها گفت دست حمار کوتاه و به مسئولان شبکه ایکس گفت جلو چشم شما این اتفاق میافتد و هویت دارندگان اکانت شما هیچگونه امنیتی ندارند.

https://t.co/P5Yqo4qkYG

ID of the similar user: 786738726466707460

Tweet Text:

جاودانگی و درگذشت قهرمان مجاهد خلق محسن نگهبان پس از ۴۴سال مبارزه با دیکتاتوری آخوندی در صفوف مجاهدین و ارتش آزادی
https://t.co/F96Cbvm96m #ایران #Iran #شیراز

ID of the similar user: 2318855006

Tweet Text:

کنفرانس در برلین- #حقوق_بشر در #ایران و صلح در منطقه، دو روی یک سکه

سیاست سکوت و بی‌عملی غرب، رژیم آخوندی را قادر کرده که فجایع بیش‌تری به بار بیاورد.

از جمله در جریان اشغال عراق، دروازه‌های این کشور به روی نیروی تروریستی قدس باز شد.

پس از افشای سایت‌های #اتمی رژیم توسط مقاومت ایران در سال ۱۳۸۱، به‌جای تحریم‌های شدید و ممنوع کردن غنی‌سازی اورانیوم، به مذاکره و امتیاز دادن به رژیم روآوردند.

Explanations:

Obviously, all the results are consistent with the taste of the user and the top similar users are chosen correctly. Therefore, looking at recommended tweets meticulously, we can conclude the recommender system has a high accuracy and low bias and variance. Or in other words, low false Negatives and false positives.

Is my RWR Algorithm Online or Offline?

Explanations:

The proposed Random Walk with Restart algorithm is a completely **online** model. Because, if you add a user, whenever he replies, retweets, or quotes a tweet, we can find easily the users which he had interaction with, and then add the node of the given user (and if some of the users which the user had interaction with, didn't exist, add the node of them) to the graph (RDD), and connect that to the node of those found users with the corresponding weights for the edges. This way, we are done!

So, the model doesn't need to be determined from the scratch and using some little processes, we can reach the desired result and find similar users or items easily though.

Advantages and Disadvantages of both Algorithms

Explanations:

The RWR algorithm is based on simulating random paths on a graph of users and items, and measuring the probability of reaching a user from another user. The collaborative filtering algorithm is based on finding users who have rated items similarly, and predicting the ratings of unknown items for a user.

Some other not-mentioned Advantages and Disadvantages of both Algorithms are as below:

The advantages of the RWR algorithm are:

- It can handle large and sparse data.
- It can find users who are indirectly connected through multiple hops on the graph, which can increase the diversity and serendipity of the recommendations.
- It doesn't suffer from cold start problem as it can suggest users or items by a reasonable accuracy, even with smaller data.

The disadvantages of the RWR algorithm are:

- It may create many false positives by getting too far from the start node. (However I have controlled this in my algorithm by setting properly the restart probability as explained, but it may occur generally)
- It is hard to include side features for users or items, such as country or age, which can improve the quality of the model.
- It is influenced by the restart probability, which controls how often the random walk returns to the target user. Choosing an optimal value for this parameter can be tricky and may require experimentation.

The advantages of the collaborative filtering algorithm are:

- It can predict the ratings of unknown items for a user, which can help to rank the items by their relevance and preference.
- It can easily include side features for users or items, by using different similarity measures or matrix factorization techniques.
- It works for any kind of item and needs no feature selection.

The disadvantages of the collaborative filtering algorithm are:

- It may only find users who have rated some common items, which can limit the scope and diversity of the recommendations and have popularity bias.
- It may require a lot of data and enough users in the system to achieve good performance and find a match, as the ratings matrix can be very large and sparse.

To compare the two algorithms, we can say that:

- The RWR algorithm can handle large and sparse data more efficiently, while the collaborative filtering algorithm can handle fresh items more easily.
- The RWR algorithm works stochastic, while the collaborative filtering algorithm works deterministic.
- The RWR algorithm can provide more diverse and serendipitous recommendations, while the collaborative filtering algorithm can provide more relevant and personalized recommendations.
- The RWR algorithm can suggest similar users with one iteration without considering the visit counts to have $\mathcal{O}(1)$ complexity, while the collaborative filtering algorithm has $\mathcal{O}(1)$ complexity, no matter how you implement that.
- The RWR algorithm is harder to include side features, while the collaborative filtering algorithm is easier to include side features.
- The RWR algorithm is influenced by the restart probability, while the collaborative filtering algorithm is influenced by the similarity measure.