HW1_Q4

November 11, 2022

1 4. Google Pagerank Algorithm (10 points)

Keywords: Pagerank, Power Method

About the dataset:

DBpedia (from "DB" for "database") is a project aiming to extract structured content from the information created in the Wikipedia project. This structured information is made available on the World Wide Web. DBpedia allows users to semantically query relationships and properties of Wikipedia resources, including links to other related datasets. for more info, see: https://en.wikipedia.org/wiki/DBpedia.

We will download two files from the data respository: * The first file – **redirects_en.nt.bz2** – contains redirects link for a page. Let A redirect to B and B redirect to C. Then we will replace article A by article C wherever needed. * The second file – **page_links_en.nt.bz2** – contains pagelinks which are links within an article to other wiki article.

Note that the data is both files is a list of lines which can be split into 4 parts: * The link to first article. * Whether it is a redirect, or a pagelink. * The link to second article. * A fullstop.

Note: Any line which cannot be split into 4 parts is skipped from consideration. **Agenda**: * In this programming challenge, you will be implementing the *google pagerank algorithm* to determine the most important articles. * This will be done by computing the principal eigenvector of the article-article graph adjacency matrix. * In this challenge, you will be applying the *power method* to extract the principal eigenvector from the adjacency matrix. * Using the computed eigenvector, we can assign each article a eigenvector-centrality score. Then we can determine the most important articles.

Environment: Ensure following libraries are installed - sklearn - numpy

Also ensure that you have around **4 GB** of memory.

Note: * Run all the cells in order. * **Do not edit** the cells marked with !!DO NOT EDIT!! * Only **add your code** to cells marked with !!!! YOUR CODE HERE !!!! * Do not change variable names, and use the names which are suggested.

---

[1]:
```
# !! DO NOT EDIT !!
# imports
import pickle
from bz2 import BZ2File
```

```
import bz2
import os
from datetime import datetime
import pprint
from time import time
import numpy as np
from urllib.request import urlopen
import scipy.sparse as sparse
pp = pprint.PrettyPrinter(indent=4)
```

[2]:
```
# !! DO NOT EDIT !!
# download the dataset and store files in local

# dbpedia download urls
redirects_url = "http://downloads.dbpedia.org/3.5.1/en/redirects_en.nt.bz2"
page_links_url = "http://downloads.dbpedia.org/3.5.1/en/page_links_en.nt.bz2"

# extarct the file-names from the urls. Needed to load the files later
redirects_filename = redirects_url.rsplit("/", 1)[1] # redirects_en.nt.bz2 ~␣
 ↪59MB
page_links_filename = page_links_url.rsplit("/", 1)[1] # page_links_en.nt.bz2 ~␣
 ↪850MB

resources = [
    (redirects_url, redirects_filename),
    (page_links_url, page_links_filename),
]

# download the files
# this will take some time
for url, filename in resources:
    if not os.path.exists(filename):
        print("Downloading data from '%s', please wait..." % url)
        opener = urlopen(url)
        open(filename, "wb").write(opener.read())
        print()
```

[3]:
```
# !! DO NOT EDIT !!
# load the data from the downloaded files
# this may take some time

#read redirects file
redirects_file = bz2.open(redirects_filename, mode='rt')
redirects_data = redirects_file.readlines()
redirects_file.close()

# pagelinks data has 119M entries
```

```
# We will only consider the first 5M for this question
pagelinks_file = bz2.open(page_links_filename, mode='rt')
pagelinks_data = [next(pagelinks_file) for x in range(5000000)]
pagelinks_file.close()
```

[4]:
```
# !! DO NOT EDIT !!
# look at the size of the data and some examples
print ('The number of entries in redirects:', len(redirects_data))
print ('A couple of examples from redirects:')
print (redirects_data[10000:10002])
print ('\n')

print ('The number of entries in pagelinks:', len(pagelinks_data))
print ('A couple of examples from pagelinks:')
print (pagelinks_data[100000:100002])
```

```
The number of entries in redirects: 4082533
A couple of examples from redirects:
['<http://dbpedia.org/resource/Proper_superset>
<http://dbpedia.org/property/redirect> <http://dbpedia.org/resource/Subset>
.\n', '<http://dbpedia.org/resource/Jean_Paul_Sartre>
<http://dbpedia.org/property/redirect> <http://dbpedia.org/resource/Jean-
Paul_Sartre> .\n']


The number of entries in pagelinks: 5000000
A couple of examples from pagelinks:
['<http://dbpedia.org/resource/Antipope> <http://dbpedia.org/property/wikilink>
<http://dbpedia.org/resource/Council_of_Constance> .\n',
'<http://dbpedia.org/resource/Antipope> <http://dbpedia.org/property/wikilink>
<http://dbpedia.org/resource/Pope_Alexander_V> .\n']
```

Note: It is worth noting here that each article is uniquely represented by its URL, or rather, the last segment of its URL

1.0.1 (a) Define a function `get_article_name` which takes as input the URL string, and extracts the last segment of the URL, which we can call as article name. (1 point)

[5]:
```
def get_article_name(url):
    name = url.split('/')[-1][:-1]
    return name
```

[6]:
```
# !! DO NOT EDIT !!
# some unit tests to validate your solution
```

```
assert get_article_name('<http://dbpedia.org/resource/Pope_Alexander_V>') ==␣
 ↪'Pope_Alexander_V'
assert get_article_name('<http://dbpedia.org/resource/Jean-Paul_Sartre>') ==␣
 ↪'Jean-Paul_Sartre'
```

1.0.2 (b) Define a function `resolve_redirects` which takes as input a list of redirect lines, and returns a map between the initial and the resolved redirect page. (2 points)

Note: Remember to ignore malformed lines which are those which do not split in 4 parts.

[7]:
```
def resolve_redirects(urls):

    # Parse urls
    redirects = {}
    for url in urls:

        url = url.split(' ')

        # Ignore bad lines
        if len(url) != 4:
            continue

        from_url = get_article_name(url[0])
        to_url =  get_article_name(url[2])

        # Skip self redirects (A -> A)
        #   or inverted duplicates (A -> B and B -> A)
        if from_url == to_url or to_url in redirects.keys():
            continue

        redirects[from_url] = to_url

    # Lookup faster in set than list
    keys_unique = set(redirects.keys())

    # Resolve
    for k, v in redirects.items():

        another_link = True
        k_hist = [k]

        while another_link:

            if v in keys_unique:
```

```
                # Link found
                k_hist.append(v)
                v = redirects[v]
            else:
                # Propogate solution back through all links
                for kh in k_hist:
                    redirects[kh] = v

                another_link = False

    return redirects
```

[8]:
```
# !! DO NOT EDIT !!
# some unit tests to validate your solution
test_input = ['<http://dbpedia.org/resource/A> <http://dbpedia.org/property/
 ↪redirect> <http://dbpedia.org/resource/B> .\n',
              '<http://dbpedia.org/resource/B> <http://dbpedia.org/property/
 ↪redirect> <http://dbpedia.org/resource/C> .\n',
              '<http://dbpedia.org/resource/C> <http://dbpedia.org/property/
 ↪redirect> <http://dbpedia.org/resource/D> .\n',
              '<http://dbpedia.org/resource/X> <http://dbpedia.org/property/
 ↪redirect> <http://dbpedia.org/resource/Z> .\n']

test_output = {'A': 'D', 'B': 'D', 'C': 'D', 'X': 'Z'}

assert resolve_redirects(test_input) == test_output
```

1.0.3 (c) Create article-article adjacency matrix.

1.0.4 Let the number of articles n. The adjacency matrix should have a value $A[i][j] = 1$ if there is a link from i to j. Note that the matrix may not be symmetric. This matrix would have rows as source, and columns as destinations. However, for further sections, we need it the other way round. Therefore, return $A^\top$ matrix.

1.0.5 Create a function `make_adjacency_matrix` that takes as input the resolved redirect map from part (b), and the list from `pagelinks_data`.

1.0.6 Return a tuple of `(index_map, A')`, where `index_map` is a map of each article to a unique number between 0 and $n-1$, also its unique numerical id. `A` is the adjacency matrix in scipy.sparse.csr_matrix format. `A'` is the transpose of matrix `A`. (2 points)

Note: Take care that if A redirects to D and X redirects to Y, and there is a pagelink entry from A to X, then the resolved pagelink entry should be D to Y.

[9]:
```
from scipy.sparse import csr_matrix
from tqdm.notebook import tqdm
```

```
def make_adjacency_matrix(redirects, pagelinks):

    # Initialize index map from redirects
    index_map = {}
    i=0
    for k, v in redirects.items():
        if v not in index_map.keys():
            index_map[v] = i
            i += 1

    # Add pagelinks to index map
    redirects_pagelinks = resolve_redirects(pagelinks)

    for k, v in redirects_pagelinks.items():
        for j in [k, v]:

            if j in redirects.keys() or j in redirects.items() or j in␣
 ↪index_map.keys():
                continue

            index_map[j] = i
            i+=1

    # Create adjaceny matrix
    keys_unique = set(redirects.keys())

    A = csr_matrix((len(index_map), len(index_map)), dtype=int)

    rp = list(redirects_pagelinks.items())

    for k, v in tqdm(rp, total=len(rp)):

        if k in keys_unique:
            source = redirects[k]
        else:
            source = k

        if v in keys_unique:
            dest = redirects[v]
        else:
            dest = v

        # [i, j] indices are flipped to go straight to transpose
        A[index_map[dest], index_map[source]] = 1
```

```
    return (index_map, A)
```

[10]:
```
# !! DO NOT EDIT !!
# some unit tests to validate your solution
test_redirects = {'A': 'D', 'B': 'D', 'C': 'D', 'X': 'Z', 'K':'L', 'M':'N'}
test_pagelinks_data = ['<http://dbpedia.org/resource/A> <http://dbpedia.org/
 ↪property/wikilink> <http://dbpedia.org/resource/X> .\n',
                       '<http://dbpedia.org/resource/L> <http://dbpedia.org/
 ↪property/wikilink> <http://dbpedia.org/resource/N> .\n',
                       '<http://dbpedia.org/resource/P> <http://dbpedia.org/
 ↪property/wikilink> <http://dbpedia.org/resource/Q> .\n']

test_output_index_map = {'D': 0, 'Z': 1, 'L': 2, 'N': 3, 'P': 4, 'Q': 5}
test_output_adjacency_matrix = np.array([[0., 1., 0., 0., 0., 0.],
                                         [0., 0., 0., 0., 0., 0.],
                                         [0., 0., 0., 1., 0., 0.],
                                         [0., 0., 0., 0., 0., 0.],
                                         [0., 0., 0., 0., 0., 1.],
                                         [0., 0., 0., 0., 0., 0.]])

output_index_map, output_A = make_adjacency_matrix(test_redirects,␣
 ↪test_pagelinks_data)

assert output_index_map == test_output_index_map
np.testing.assert_array_equal(output_A.toarray(), test_output_adjacency_matrix.
 ↪T)
```

```
  0%|          | 0/3 [00:00<?, ?it/s]
```

```
/home/rphammonds/projects/dsc210/.env/lib/python3.10/site-
packages/scipy/sparse/_index.py:103: SparseEfficiencyWarning: Changing the
sparsity structure of a csr_matrix is expensive. lil_matrix is more efficient.
  self._set_intXint(row, col, x.flat[0])
```

1.0.7 (d) Apply the above functions on the dataset to create adjacency matrix A and other relevant maps as directed below. Then apply SVD from sklearn on the adjacency matrix to determine principal singular vectors. (2 points)

[11]:
```
# 1. with redirects_data as input, use the resolve_redirects function to␣
 ↪generate the redirects_map
redirects_map = resolve_redirects(redirects_data)

# 2. with redirects map from previous step pagelinks_data as inputs,
#    use the make_adjacency_matrix to generate index_map and adjacency_matrix
index_map, X = make_adjacency_matrix(redirects_map, pagelinks_data)
```

```
# 3. using index_map, create a reverse_index_map, which has the article name as␣
 ↪key, and its index as value
reverse_index_map = {v:k for k, v in index_map.items()}
```

```
  0%|          | 0/46180 [00:00<?, ?it/s]
```

[12]:
```
# !! DO NOT EDIT !!
# Now we will save the csr matrix, index_map and reverse_index_map in pickle␣
 ↪files
# so that we do not have to recompute steps (a)-(d) next time we load the␣
 ↪program
# (Note: beneficial only when working on local machine, as colab session times␣
 ↪out)
PATH='./'
pickle.dump(X, open(PATH+'X.pkl', 'wb'))
pickle.dump(index_map, open(PATH+'index_map.pkl', 'wb'))
pickle.dump(reverse_index_map, open(PATH+'reverse_index_map.pkl', 'wb'))

# free up RAM
del(redirects_data, pagelinks_data)
```

! ———- Checkpoint ——— !

[13]:
```
# !! DO NOT EDIT !!
# Load the data from here
PATH='./'
X = pickle.load(open(PATH+'X.pkl', 'rb'))
index_map = pickle.load(open(PATH+'index_map.pkl', 'rb'))
reverse_index_map =  pickle.load(open(PATH+'reverse_index_map.pkl', 'rb'))
```

Apply `randomized_svd` from sklearn on the adjacency matrix. Extract top 5 components and run for 3 iterations.

[14]:
```
from sklearn.utils.extmath import randomized_svd

U, s, V = randomized_svd(X, 5, n_iter=3)
```

```
/home/rphammonds/projects/dsc210/.env/lib/python3.10/site-
packages/sklearn/utils/extmath.py:370: FutureWarning: If 'random_state' is not
supplied, the current default is to use 0 as a fixed seed. This will change to
None in version 1.2 leading to non-deterministic results that better reflect
nature of the randomized_svd solver. If you want to silence this warning, set
'random_state' to an integer seed or to None explicitly depending if you want
your code to be deterministic or not.
  warnings.warn(
```

[15]:
```
# !! DO NOT EDIT !!
# now, we print the names of the wikipedia related strongest components of the
```

```
# principal singular vector which should be similar to the highest eigenvector
print("Top wikipedia pages according to principal singular vectors")
pp.pprint([reverse_index_map[i] for i in np.abs(U.T[0]).argsort()[-10:]])
pp.pprint([reverse_index_map[i] for i in np.abs(V[0]).argsort()[-10:]])
```

```
Top wikipedia pages according to principal singular vectors
[   'Mathematics_%28disambiguation%29',
    'Population_density.png',
    'Roman_Emperor',
    'Catholic_Church',
    'Association_to_Advance_Collegiate_Schools_of_Business',
    'Japan_Self-Defense_Forces',
    'Snowboarding_pictogram.svg',
    'Rhodesian_Bush_War',
    'CNN_Center',
    'Cinema_of_the_United_States']
[   'Muckleshoot',
    'Cinema_of_the_United_States',
    'Robert_Fulton',
    'North_American_Free_Trade_Agreement',
    'Thora_Birch',
    'List_of_business_schools_in_the_United_States',
    'Military_of_The_Gambia',
    'Snowboarding_at_the_2002_Winter_Olympics',
    'History_of_Zimbabwe',
    'CNN']
```

1.0.8 (e) The pagerank algorithm

1.0.9 In this final section we will implementing the google pagerank algorithm by computing principal eigenvector using power iteration method. (3 points)

1.0.10 To start with the power iteration method, we first need to make the matrix X obtained in (d) *column stochastic*. A column stochastic matrix is a matrix in which each element represents a probability and the sum each column adds up to 1. Recall that X is a matrix where the rows represent the destination and columns represents the source. The probability of visiting any destination from the source s is $1/k$, where k is the total number of outgoing links from s. Use this information to make the matrix column stochastic.

[ ]:
```
#######
# !!!! YOUR CODE HERE !!!!

#######
```

1.0.11 ***Dangling Nodes*: There may exisit some pages which have no outgoing links. These are called as dangling nodes. If a random surfer just follows outgoing page links, then such a person can never leave a dangling node. We cannot just skip such a node, as there may be many pages pointing to this page, and could therefore be important.**

1.0.12 **To solve this problem, we introduce teleportation which says that a random surfer will visit an outgoing link with β probability and can randomly jump to some other page with a $(1-\beta)/n$ probability (like through bookmarks, directly going through URL, etc.). Here n is the total number of pages under consideration, and β is called the damping factor. So now, the modified transition matrix is:**

1.0.13 $R = \beta X + \frac{(1-\beta)}{n} I_{n \times n}$

1.0.14 **where X is the column stochastic matrix from previous step, and $I_{n \times n}$ is a $n \times n$ identity matrix.**

1.0.15 **Using the transition matrix R, use the power iteration method to solve for the principal eigenvector $\mathbf{p}_{n \times 1}$. Start with an initial guess of $\mathbf{p}_{n \times 1} = [\frac{1}{n}, \frac{1}{n}, ..., \frac{1}{n}]$, which intuitively represents that a random surfer can start at any page with a $\frac{1}{n}$ probability. Use a damping factor of 0.85, and perform a maximum of 100 iterations.**

1.0.16 **Report the top 10 page names which correspond to the top 10 scores (magnitudes) in the principal eigenvector.**

[ ]:
```
#######
# !!!! YOUR CODE HERE !!!!

#######
```
