Name: Sharon (Wenxin) Zhang
Andrew ID: wenxinz3

Machine Learning for Text and Graph-based Mining

Homework 1 - Template

1. Statement of Assurance

I certify that all the materials and codes I submitted are the original work done only by myself.

2. Experiments

a) Describe the custom weighing scheme that you have implemented. Explain your motivation for creating this weighting scheme.

For the custom weighing scheme, I apply the log(base e) function to the PageRank(PR) score and apply 0.1-0.9 weighting to PR-IR (information retrieval) score.

The rationale comes from observing the PageRank scores produced by all three algorithms. Scores from all three algorithms are positive but extremely small, compare to the provided IR scores that are all negative and large. Without any scaling, the PageRank scores would be relatively insignificant compared to the IR scores. Any kind of non-zero weighting would not significantly change the final combine scores (the ranking could change a lot though), unless we ignore the IR scores (which is the NS method).

To convert the PageRank scores into a more comparable scale, log function makes the most sense to me, especially when we have input values that are positive but in decimal. The resulting values after log would become large in absolute value but negative, which is more comparable to the format of IR scores. Additionally, log function is strictly increasing and hence would preserve the order of values.

The 0.1-0.9 weighing comes from experiment. I start with half-half weighing but the MAP and recall values aren't delightful for TSPR algorithm - much lower than WS method. This is likely due to the log function that scales the PageRank scores, which make both scores at an almost-equal scale. Given the similarly level of scale, if we place equal weight to both scores, we consider the PageRank and IR score equally important. However, since IR score is sensitive to the input query, but PageRank scores are calculated offline, placing a heavier weight on IR score could preserve the sensitivity to the query, which likely improve the search accuracy.

After several experiments on 0.3-0.7, 0.2-0.8, the comparably best performance is reached by 0.1-0.9 weighing.

b) Report of the performance of the 9 approaches.

1. Metric: MAP

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0452	0.2636	0.2635
QTSPR	0.0483	0.2636	0.2542
PTSPR	0.0515	0.2637	0.2547

2. Metric: Precision at 11 standard recall levels

P@0

Method \ Weighting Scheme	NS	WS	CM
GPR	0.1669	0.8405	0.8405
QTSPR	0.1779	0.8405	0.8206
PTSPR	0.2199	0.8405	0.8345

P@10

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0834	0.5926	0.5926
QTSPR	0.0914	0.5926	0.5753
PTSPR	0.1042	0.5926	0.5742

P@20

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0784	0.4732	0.4732
QTSPR	0.0808	0.4731	0.4741
PTSPR	0.0861	0.4731	0.4635

P@30

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0737	0.3781	0.3781
QTSPR	0.0747	0.3780	0.3675
PTSPR	0.0750	0.3780	0.3656

P@40

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0697	0.3145	0.3145
QTSPR	0.0683	0.3148	0.3040
PTSPR	0.0716	0.3148	0.2989

P@50

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0651	0.2430	0.2426
QTSPR	0.0616	0.2428	0.2314
PTSPR	0.0637	0.2433	0.2295

P@60

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0530	0.1677	0.1673
QTSPR	0.0497	0.1673	0.1570
PTSPR	0.0503	0.1677	0.1603

P@70

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0300	0.0915	0.0915
QTSPR	0.0275	0.0915	0.0888
PTSPR	0.0277	0.0915	0.0902

P@80

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0114	0.0550	0.0551
QTSPR	0.0116	0.0550	0.0557
PTSPR	0.0115	0.0550	0.0560

P@90

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0074	0.0388	0.0388
QTSPR	0.0071	0.0388	0.0365
PTSPR	0.0072	0.0388	0.0371

P@100

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0041	0.0101	0.0101
QTSPR	0.0040	0.0101	0.0078
PTSPR	0.0040	0.0101	0.0082

3. Metric: Wall-clock running time in seconds

Note: PageRank running time stays the same for the same algorithm, regardless of the weighting schemes. Below chart displays running time in format of "PageRank | Retrieval".

Method \ Weighting Scheme	NS	WS	CM
GPR	0.636 | 0.00522	0.636 | 0.00542	0.636 | 0.00551
QTSPR	3.223 | 0.00556	3.223 | 0.00610	3.223 | 0.00673
PTSPR	3.197 | 0.00835	3.197 | 0.00517	3.197 | 0.00696

4. Parameters

* For TSPR, beta = 0.15, gamma = 0.05

  * *Idea is to place heavier weight on the topic probability distribution at 3:1 ratio.*

* Converge Threshold: 10^(-8).

* WS method: 0.2 for IR, 0.8 for PageRank

* CM method: 0.9 for IR, 0.1 for log(PageRank)

c) Compare these 9 approaches based on the various metrics described above.

MAP

When looking vertically across algorithms, MAP scores do not vary too much for different PageRank algorithms. PTSPR tends to be slightly better than QTSPR, which is slightly better than GPR. When looking horizontally across weighting methods, both WS and CM performs much better than NS. This is likely because the incorporation of IR scores, which makes the final scores sensitive to the query and might be more effective. The differences between WS and CM are not significant.

P@Recall Levels

In general, for all three algorithms, the P@n value decrease with the recall precision increase. It's interesting that at low recall levels, GPR has worse precision under NS method compared to TSPR, but better precision under CM method compared to TSPR. The difference is not very significant though. As the recall level passes 50%, GPR most often yields slightly better metrics under all three weighing schemes.

At any given P@n and for any algorithms, the CM and WS weighing schemes yield much better results than NS method.

Within a given recall level, PTSPR tends to have slightly better precision than QTSPR, particularly under CM weighing method.

Under the WS method, for a given recall level, all three algorithms have very close precision.

Running Time

GPR absolutely outperform in running time. To calculate PageRank scores, QTSPR and PTSPR take about 50 times more than GPR takes. On the iteration number, it takes 10 iterations for GPR to converge, but on average 65 iterations for TSPR to converge.

The retrieval time are much less significant compared to PageRank running time, but PTSPR has comparatively longer running time and GPR is still the quickest. The difference in retrieval runtime caused by different weighing schemes do not have an obvious pattern. For PTSPR, the NS weighing cost the most, but for GPR and QTSPR, the CM weighing cost the most.

d) Analyze these various algorithms, parameters, and discuss your general observations about using PageRank algorithms.

When consider purely the PageRank scores, GPR is less accurate than TSPR as it does not take user/query into account. This could be seen from the MAP under NS weighing, where the IR scores are not included. PTSPR has the best MAP, and better P@n at low recall level. Hence, the document-topic distribution could be quite important to a more accurate PageRank model.

When incorporating IR scores, however, GPR tends to performance slightly better than TSPR, or at a similar level. This is a bit surprising to me, as I would expect the TSPR to continue outperform. However, the inclusion of IR scores might become the leading factor in the final scores, which could be one reason that the MAP scores do not differ too much across three PageRank algorithms.

The weighing methods could be effective to the final accuracy, as it tries to find a better way to keep the sensitivity to query but also add user/query features to the scores. Overall, the WS method has the best performance regardless of algorithms.

Lastly, in terms of the running time, GPR is the most efficient. If the input data is significantly large, GPR might be a better choice in terms of time efficiency.

e) **1. What could be some novel ways for search engines to estimate whether a query can benefit from personalization?**

From query's perspective, if such type of query usually benefits from personalization regardless of users, then the query has higher probability of benefiting from personalization. We could conduct experiments on different types of queries, categorized by features including query length, query topic, query term POS and etc., run the GPR and PTSPR on each query, and compare the accuracy metrics. If some categories of query tend to always perform better with PTSPR while others do not, we might consider queries of these types likely benefit from personalization.

From user's perspective, we could store users' search history to derive the users' interest of topics and measure the similarity among users. Then for a given user, if a query has been searched by his/her similar users, it's likely that the query could benefit from personalization.

2. What could be some novel ways of identifying the user's interests (e.g. the user's topical interest distribution Pr(t|u)) in general?

In addition to gathering user's historical searches and calculate the empirical distribution of topics, we could also:

a. For each search that the user click-through, look at the out-links of that page and analyze what topics are the out-links tight to, and give some weight on these indirectly related topics

b. Gather user's search date and time – for search topics that appear frequently and consistently (say five searches every month within a year, total 60 searches), give more weight to them compared to topics appearing only in five days within a year, but each day 12 searches. The compact search in short time frame might indicate the search result isn't accurate enough, or the user is just temporarily interested in the topic.
c. Within a specified timeframe, say 2 minutes, we assume all the searches by a user is targeting this same information needed. Then for all search queries, we collect the keywords and pick the words with highest frequently, which is likely the main thesis of the search. For topics related to these top words, give higher weight.
d. Use collaborative filtering methods or clustering to group users with similar search keywords, compare their search topic profiles, and assign higher weight to the topics that the similar users have searched but the target user has not yet.

3. Details of the software implementation

a) **Describe your design decisions and high-level software architecture;**

The general idea of my design is to separate the calculation of PageRank scores from the calculation of IR scores (provided), and within the PageRank scores, for TSPR, separate the calculation of link-based scores from the calculation of content-based scores (provided).

I create two files, one for all helper and calculation functions (link_analysis.py), and one for running the entire PageRank calculation across three algorithms across three weighing schemes and save the resulting scores into 9 different txt files (link_analysis_runner.py).

Inside link_analysis.py, I created two functions for calculating PageRank scores at convergence, one for GPR(global_PR) and one for link-based scores in TSPR (link_score_TSPR). There isn't content-based score concept for GPR. For both PageRank functions, the idea is to iterate the input r vector until convergence, using the formula $(1-\alpha)M^T r + \alpha p_0$ for GPR and $\alpha M^T r + \beta p_t + \gamma p_0$ for TSPR. Note that the vector returned by link_score_TSPR is topic based. For initialization, I set $r_0,\ p_0 = (\frac{1}{n},\ \frac{1}{n},\ \frac{1}{n}, \dots,\ \frac{1}{n})$.

For TSPR, on top of the link-based score, I created the function full_score_TSPR to add content-based scores. The function is design to iterate throw all the topics, calculated the final PageRank scores for all topics, and combine the PageRank scores per topic into a $n\ x\ T$ matrix. Then, the PageRank matrix is multiplied by the query- or user-topic distribution (content-based scores) to become the resulting matrix, which is a $n\ x\ Q$ matrix, where Q is the number of unique user-query in consideration. Instead of calculating PageRank scores per user-query, my

function calculation all PageRank scores for all user-query at one place. However, this would require a $n\ x\ Q$ matrix to store the values, which might not be the most space efficient.

The next step is combining PageRank scores to IR scores, which is done via function "combine_IR_PR_per_query_user". This function is specific to user-query input, and subject to the weighing schemes.

Finally, there are two aggregate functions, one for GPR (run_GPR_into_file) and one for TSPR (run_TSPR_into_file), that combine the two (three for TSPR) steps from above, and write the results into txt files.

NOTE: a special design of my program is that, instead of using the dense matrix M directly, I decompose M into a sparse matrix "trans_mtx" that only contains values from the given transition.txt, adding normalization, and a $n\ x\ 1$ vector "zero_outlink_vector" that each cell has identical values, and the values would be updated in each iteration. The initial value of this vector is calculated by the number of documents without any out links divided by total number of documents, and average across all documents - so divided by n^2. By such decomposition, my function takes only a sparse matrix rather than the dense M, and a dense vector that would be updated in each iteration. This decomposition helps with the implementation and facilitates the run time, since carrying a dense matrix is both time and space inefficient.

The decomposition is identical to the dense transition probability matrix M, as $M = M_{sparse} + M_{zero\_outlinks}$. M_{sparse} remains in the formula, and $M_{zero\_outlinks}$ is a $n\ x\ n$ matrix with 0 for all rows that have values in M_{sparse}, and $\frac{1}{n}$ for all rows without any values in M_{sparse}. Then we have $M^T r = {M_{sparse}}^T r + {M_{zero\_outlinks}}^T r$. The result of ${M_{zero\_outlinks}}^T r$ is just a $n\ x\ 1$ matrix, each cell has identical value, which is the sum of $\frac{1}{n}$ for all documents without out-links.

Since I initialize $r_0 = (\frac{1}{n}, \frac{1}{n}, \frac{1}{n}, \dots, \frac{1}{n})$, the starting value for zero_outlink_vec would be $\frac{\#doc\ without\ any\ outlinks}{n^2}$. As the iteration starts, zero_outlink_vec must be updated with the new $\#doc\ without\ any\ outlinks$.

b) **Describe major data structures and any other data structures you used for speeding up the computation of PageRank;**

Scipy.COO: for storing the sparse transition matrix *(speedy for matrix calculation)*
Scipy.CSC: for storing the doc-topic probability matrix *(speedy for accessing column and dot product)*

Pandas.DataFrame: for storing topic-query and topic-user distribution matrix, all provided IR scores, resulting final GPR and TSPR scores per user-query *(used only for table with smaller size, easy to write into txt and apply column level calculation)*
Numpy.ndarray: TSPR PageRank score only matrix, GPR PageRank score only vector *(speedy for vector calculation, transpose, and column accessing)*
Numpy.vstack: r vector and zero_outlink_vector *(speedy for vector calculation)*

c) **Describe any programming tools or libraries and programming environment used;**

I code in Python3 using PyCharm Community. My code import pandas, numpy, scipy, time, and os. The machine I used is MacAir M2 2022 with macOS Monterey system and the version is 12.4.

The DataFrame structure under pandas is used for storing provided search scores, calculated page scores, and adjusted scores. Function **.to_csv()** is used to write into txt file.

The CSC and COO matrix structures under scipy are used for storing transition and distribution matrix and implementing matrix multiplication. Function **.transpose()** is used for matrix transpose.

The vstack and ndarray structures under numpy are used for storing r_0, p_0 and other updated $r_i$vectors, and implementing **.dot()** and **.transpose().**

Package os and time are imported for calculating running time and accessing file by path.

d) **Describe strengths and weaknesses of your design, and any problems that your system encountered**

Strengths:

- Decompose the dense transition matrix into sparse matrix and one $n\ x\ 1$ vector, which save the memory space while preserving the matrix values.
- Construct a $n\ x\ T$ matrix to store the link-based TSPR for each topic, and hence calculate the combined TSPR PageRank scores at once for all query-user pairs rather than one by one, by matrix multiplication with $T\ x\ Q$ matrix storing content-based probabilities.
- Store the map between query-user index to queryID-userID pair into a dataFrame so as to make sure the index accessing would be corrected.
- Clear sequence on score calculation, from PageRank linked-based score to PageRank combined score (adding content-based for TSPR) to GPR/TSPR final score (adding IR).

Weaknesses:

- The TSPR PageRank matrix takes memory space to store, same as the combined IR score dataFrame for all query-user IR score. My code does not trigger any memory issue, but with a large n, T and Q, the issue could arise.
- Various data structures are used in my design for efficient purpose, but could make the code itself less consistent to read. Potential improvement includes replace the use of DataFram by Numpy structure to keep consistency.