Name: Joao Miguel De Almeida Vares Coelho
Andrew ID: jmcoelho

Machine Learning for Text and Graph-based Mining

Homework 1

1. Statement of Assurance

I certify that all the submitted material is original work and was done by me.

2. Experiments

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

Looking ate the output scores by both ranking models, the magnitude and range of the values are very different. The retrieval scores seem to be log-probability values, ranging in]-inf, 0[, while the page rank score is a probability value ranging in [0,1]. This way, I tried custom schemes that aimed to balance this difference:

- (WRRF) Weighted reciprocal rank fusion:
  - $\text{WRRF\_Score(d)} = \frac{(1-\alpha)}{k + PR\_Rank(d)} + \frac{\alpha}{k + SS\_Rank(d)}$
  - In the previous equation, PR_Rank(d) is the ranked position of document d given the page rank scores, and SS_Rank(d) is the ranked position of document d given the search scores. The constant k was set to 60. By leveraging the ranks instead of the scores, the metric avoids the disparity of magnitude and range.
- (WSLP) Weighted sum of log probabilities:
  - SLL_Score(d) = $(1-\alpha)\log PR\_Score(d) + \alpha S\_Score(d)$
  - In the previous equation, PR_Score(d) is the pagerank score for document d, and S_Score(d) is the search score for document d. Since S_Score(d) is already a log-probability, it is used as-is. By taking the sum of two log probabilities, it is equivalent to the log of the product of the original probabilities. A score given by the product can be seen as a proxy for the logical AND operator, i.e., give high scores when both components are high.

The values of constant α are discussed in the Parameter section. The usage of weighted schemes favoring the search scores is due to the poor performance of the page rank scores in this specific dataset, since the labels are most likely rating relevance, while page rank evaluated popularity.

I thought it was important to share both options as they are conceptually different. Nevertheless, the one that achieved the best results was WRRF, which is the one the program runs, and for which the results are available below.

b) Report of the performance of the 9 approaches.

1. Metric: MAP

Method \ Weighting Scheme	NS	WS	CM
GPR	0.460	0.2636	0.2636
QTSPR	0.455	0.2635	0.2636
PTSPR	0.454	0.2635	0.2636

2. Metric: Precision at 0%

Method \ Weighting Scheme	NS	WS	CM
GPR	0.1455	0.8405	0.8405
QTSPR	0.1587	0.8405	0.8405
PTSPR	0.1537	0.8405	0.8405

3. Metric: Precision at 10%R

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0877	0.5926	0.5926
QTSPR	0.0829	0.5926	0.5926
PTSPR	0.0829	0.5926	0.5926

4. Metric: Precision at 20%R

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0784	0.4732	0.4732
QTSPR	0.0746	0.4732	0.4731
PTSPR	0.0746	0.4732	0.4731

5. Metric: Precision at 30%R

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0740	0.3781	0.3781
QTSPR	0.0707	0.3781	0.3777
PTSPR	0.070	0.3781	0.3777

6. Metric: Precision at 40%R

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0697	0.3145	0.3144
QTSPR	0.0668	0.3144	0.3147
PTSPR	0.0668	0.3145	0.3147

7. Metric: Precision at 50%R

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0651	0.2430	0.2430
QTSPR	0.0617	0.2426	0.2426
PTSPR	0.0617	0.2426	0.2426

8. Metric: Precision at 60%R

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0532	0.1677	0.1677
QTSPR	0.0500	0.1673	0.1673
PTSPR	0.0500	0.1673	0.1673

9. Metric: Precision at 70%R

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0295	0.0915	0.0915
QTSPR	0.0270	0.0915	0.0915
PTSPR	0.0270	0.0915	0.0915

10. Metric: Precision at 80%R

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0113	0.0551	0.0550
QTSPR	0.0113	0.0550	0.0550
PTSPR	0.0133	0.0550	0.0550

11. Metric: Precision at 90%R

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0075	0.0338	0.0338
QTSPR	0.0074	0.0338	0.0338
PTSPR	0.0074	0.0338	0.0338

12. Metric: Precision at 100%R

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

13. Metric: Wall-clock running time in seconds (RETRIEVAL)

Method \ Weighting Scheme	NS	WS	CM
GPR	0.0361	0.0376	0.0362
QTSPR	0.0405	0.0405	0.0402
PTSPR	0.0409	0.0404	0.0425

14. Metric: Wall-clock running time in seconds (POWER ITERATION)

Method \ Weighting Scheme	NS	WS	CM
GPR	69.1731		
QTSPR	71.8727		
PTSPR			

15. Metric: Wall-clock running time in seconds (NORMALIZATION)

Method \ Weighting Scheme	NS	WS	CM
GPR	329.6547		
QTSPR			
PTSPR			

Estimated run-time of the script, computing everything requested:

$$329.6547 + 69.1731 + 71.8727 + 9 \times 38 \times 0.0400 = 484.3805\text{s}$$

16. Parameters

- GPR: 80% for transition matrix, 20% for teleport vector, as requested.
- QTSPR: 80% for transition matrix, 5% for teleport vector, 15% for topic vector.
- PTSPR: 80% for transition matrix, 5% for teleport vector, 15% for topic vector.
- WS: 60% for search score, 40% for pagerank score.
- CM: 90% for search score, 10% for pagerank score.

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

Ultimately, there is very little difference among the approaches. For comparison, the performance of the search-only system would have a MAP of 0.2635. So, the introduction of PageRank as a popularity metric brings few benefits for this specific dataset, as no significant performance increase was noted. Some interesting insights:

- QTSPR and PTSPR did not manage to outperform the vanilla GPR, except at precision at 0% recall for the NS approach. This means that these approaches were slightly better than the GPR when selecting the top-1 result, but have a more segmented list of results elsewhere since the MAPs are lower.
- QTSPR and PTSPR and very similar. Hence, the user-topic distribution and query-topic distributions probably show similar behaviors.

- The NS results alone are very low, solidifying that pagerank values are not very useful for this dataset, since the labels are most likely evaluating relevance only, not taking popularity into account.
- Personally, I could not find a CM that would greatly improve over the WS. The chosen one obtains comparable results.

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

The algorithms share multiple similarities. Global pagerank yields one final vector of scores, while topic sensitive pagerank yields one per topic, which are then weighted by either a query-topic distribution or user-topic distribution. Run times are fairly similar for both GPR and TSPR. The parameters were used as asked in the statement: 80% of following the transition matrix, which leaves 20% for the teleportation vector. In the case of the TSPR, 15% went to the topic teleportation vector, and 5% to the global teleportation vector. Changing the assignment of the 15%/5% to other values did no yield significant changes in the final score. Note that the value associated with the teleportation vector with the uniform distribution was always kept above 0, otherwise the process would not be Markovian.

The normalization step, specifically the part where zero-rows need to be normalized, showed itself to be very important. Without it, the process is no longer Markovian. Empirical tests showed that without normalizing the zero-rows, both algorithms would converge (wrongly) in less than 10 seconds due to sink nodes.

It is also worth noting that since the computation of pagerank vectors only needs to be done once, the algorithms offer a fast run-time experience to a possible users. Updates to the vectors can be done periodically and offline.

Regarding the weighting schemes, all of them ended up favoring the search scores. It was the only way to obtain comparable results. This further solidifies the little importance pagerank values have to add to the ranking process in the dataset.

Overall, pagerank-based algorithms are very useful to identify authorities in a linked structure. In this case it did not help, but it is not hard to think of examples where it would. For instance, outside of the web search domain: Given the task of key-phrase extraction from a single document, one could build a transition matrix based on n-gram overlap, with a teleportation vector following a distribution given by n-gram similarity and idf values over a collection. The process would converge on the most popular words of the document, which can be used as potential key-phrases.

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

- If the user is posing multiple similar queries within a short time frame, it may mean that he can't find what he needs. In that case, personalizing should work better.
- Instead of looking through a single user's history of queries, one could look into a global set of queries posed to a search engine by all users, and define which ones benefit more from personalization, e.g., through online preference testing. For instance, I'd argue that "research" queries would benefit more from personalization than simple navigational web queries. As such, identifying which types of queries benefit more from personalization could be modeled from query history and online testing, and new queries assigned a probability of needing personalization given the observed data.

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?

- User History: Assuming one has access to the user's history, we can cluster accessed documents by topic. Can then draw a distribution by topic from e.g. the density of each cluster.
- User Session Interest: History reflects all the past interests of the user. However, in a prolonged interaction with a search engine, more refined information can be drawn. Instead of relying only on the past interactions, current queries being posed can be categorized. If it follows a pattern, then the topical distribution can be weighted to prioritize the current search needs.

3. Details of the software implementation

N will be used throughout the descriptions to refer to the number of nodes.

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

Most of the solution is implemented in a python module named "pagerank". This model exposes classes which model the (G/TS) PageRank algorithm, and multiple utility functions (e.g., matrix normalization; matrix loading; weighting functions; retrieval/re-ranking functions). The classes modeling the PageRank algorithm follow the efficient computation that was introduced in class. For the global page rank, it can be written as:

$$r^{(k+1)} = (1-\alpha)M^T r^{(k)} + \alpha p_0 \ ,$$

Where M^T is the normalized transition matrix (where every row has outlink distribution), and p_0 is the teleportation vector with an uniform distribution over all nodes. Similarly, for the topic sensitive page rank, the following update was implemented:

$$r_t^{(k+1)} = \alpha M^T r_t^{(k)} + \beta p_0 + \gamma p_t$$

Both implementations follow the Power Iteration implementation, with the stopping criteria being $||r^{(k+1)} - r^{(k)}|| < 10e - 8$. The topic-specific teleportation vectors (p_t) follow the computation introduced in class.

To normalize the matrix, scikitlearn's normalize function was used. It leverages C-bindings for python, being able to normalize a sparse matrix very efficiently. However, since the implementation runs on top of sparse data structures, zero-rows are not normalized. As such, I included this behavior, by creating a matrix where the zero-rows of the original matrix are now populated with (1/(N-1)), except for the diagonal, and all other rows are empty. Then, the fully normalized matrix is given by the sum of the new matrix, and the one where the zero rows were not normalized.

The retrieval/re-ranking pipeline is implemented as follows: for a query, the top-500 document ranking from search is loaded into an array of tuples [(doc id, search score)]. Then, this array is updated to [(doc id, new score)], with the new score depending on the scoring function being used (NS, WS, or CM), and sorted by score.

The main.py file is a script which calls the above-described functions to solve the proposed exercises. It is important to note that the GPR and the TSPR computations are done only once each, and the final vectors stored in memory for efficiency (QTSPR and PTSPR share the same r_t vectors, the only difference is the weighting topic probability distribution.

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

The transition matrix was stored in a sparse matrix format (scipy.sparse.crs_matrix). This is possible because the efficient implementation was used. For instance, if I were to have combined M with a full teleportation matrix E (i.e., B=((1-a)M+aE)), this would not be possible as B would no longer be sparse. As such, since aEr = p0 where p0~Uniform(N), teleportation vectors are stored as numpy arrays. Although topic-specific teleportation vectors can be considered sparse, using a sparse array did not yield improvements, hence a normal numpy array was used as well. To support faster computation of topic sensitive pagerank, one topic vector and one teleportation vector per topic are stored in a matrix, to allow writing the iteration as a single chain matrix multiplication leveraging numpy.

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

The environment can be replicated as follows:

1. conda create -n ml4tgm-hw1 python=3.6.7
2. conda activate ml4tgm-hw1
3. pip install numpy
4. pip install scipy
5. pip install scikit-learn

Numpy was used for matrix multiplication. Scipy was used for the sparse matrix representation. Scikit-learn was used for the matrix normalization.

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

The proposed solution implements what was asked. The design is clean and extendable, and the results are as expected. Execution-time wise, the timings are OK, but it can be further optimized:

- Parallelization: There is a bottleneck on matrix normalization, which is being done sequentially. However, rows are independent. So, l1-row-wise normalization can be distributed to multiple threads to speed up. Moreover, for the TSPR computation, the computations for each r_t vector are also independent and can be distributed.
- Study on initial r_0 vector: The initial r_0 vector won't influence the outcome (as long as it is not orthogonal to the principal eigenvector of the matrix), but it does influence time (the closest the initial vector is to the principal eigenvector, the faster the convergence). A uniform initial distribution was used, but others could have been tested.

Overall, the proposed solution achieves decent run times by following the efficient implementations that were described in class and leveraging sparse data structures. The major bottleneck is in normalizing the zero-rows of the sparse matrix. This is an expensive task to perform on sparse matrices, but it is mandatory – I tested running without the zero-row normalization (normalizing only the other ones, which is fast), and the algorithm converged in less than 10 seconds to sink nodes. I tried another approach to try to mitigate this issue: before normalizing the matrix, the diagonal is set to 1, i.e., as if all nodes have some probability of jumping to themselves. This way, there are no zero rows, i.e., zero row normalization is no longer a bottleneck. The whole script runs much faster, with the power iterations running in less than one second. This leads me to believe that while we are still dealing with a Markov process, the high probability of a sink node going back to himself results in poor convergence results.