Johan Boye, KTH, 2024-02-02

DD2477 Search Engines and Information Retrieval Systems

Assignment 2: Ranked Retrieval[1]

The purpose of Assignment 2 is to learn how to implement ranked retrieval. You will learn 1) how to include tf-idf scores in the inverted index; 2) how to handle ranked retrieval from multiword queries; 3) how to use PageRank to score documents; 4) how to combine tf-idf and PageRank scoring; 5) ways of approximating cosine similarity for tf-idf computation; and 6) how to use the HITS algorithm to score documents.

The recommended reading for Assignment 2 is that of Lectures 4 and 5.

Assignment 2 is graded, with the requirements for different grades listed below. At the beginning of the oral review session, the teacher will ask you what grade you aim for and ask questions related to that grade. The assignment can only be presented once (unless you get an F) – you cannot raise your grade by doing additional tasks after the assignment has been examined and given a grade. ***Come prepared for the review session!*** *The review will take 20 minutes or less, so have everything in order.*

E: *Completed Task 2.1-2.5 with some mistakes that could be corrected at the review session.*
D: *Completed Tasks 2.1-2.5 without mistakes.*
C: *E + Completed Task 2.6*
B: *C + Completed Task 2.7*
A: *B + Completed Task 2.8*

These grades are valid for review on February 28, 2023. See the Canvas pages for the grading of delayed assignments.

Assignment 2 is intended to take around 40h to complete.

Computing Framework

For Tasks 2.1-2.2, you will be further developing your code from Assignment 1. For Tasks 2.5-2.8, you will be using a source code skeleton and some data downloadable from Canvas.

Task 2.1: Ranked Retrieval

Extend the **`search`** method in the **`Searcher`** class to implement ranked retrieval. For a given search query, compute the cosine similarity between the tf_idf vector of the query

[1] With contributions by Dmytro Kalpakchi, André Silva, Jussi Karlgren, and Hedvig Kjellström.

and the tf_idf-vectors of all matching documents. Then sort documents according to their cosine similarity score.

You will need to add code to the **`search`** method, so that when this method is called with the **`queryType`** parameter set to **`Index.RANKED_QUERY`**, the system should perform ranked retrieval. You will furthermore need to add code to the **`PostingsList`**, **`PostingsEntry`**, and **`Searcher`** classes, to compute the cosine similarity scores of the matching documents. To sort the matching documents, assign the score of each document to the **`score`** variable in the corresponding **`PostingsEntry`** object in the postings list returned from the **`search`** method. If you do this, you can then use the **`sort`** method in the built-in **`java.util.Collections`** class.

When you have finished adding to the program, compile and run it, indexing the data set **`davisWiki`**. Select the "Ranked retrieval" option in the "Search Options" menu, and try the following two search queries:

zombie	**attack**
which could result in the list	which could result in the list
Found 36 matching document(s)	**Found 228 matching document(s)**
0. JasonRifkind.f ... **1. Zombie_Walk.f ...** **2. EmilyMaas.f ...** **3. AliciaEdelman.f ...** **4. Kearney_Hall.f ...** **5. Spirit_Halloween.f ...** **6. Zombies_Reclaim_the_Streets.f ...** **7. StevenWong.f ...** **8. Measure_Z.f ...** **9. Scream.f ...** *etc.*	**0. TheWarrior.f ...** **1. Measure_Z.f ...** **2. Kearney_Hall.f ...** **3. Muilop.f ...** **4. bg-33p.f ...** **5. Furly707.f ...** **6. PamAarkes.f ...** **7. s.martin.f ...** **8. TrustInMe.f ...** **9. stevenscott.f ...** *etc.*

With one-word queries, the numbers above are equal to the length-normalized tf_idf scores of each document with respect to the query term. Our lists above were computed with a tf_idf score for document *d* and query term *t:*

- $\text{tf\_idf}_{dt} = \text{tf}_{dt} \times \text{idf}_t / \text{len}_d$
- $\text{idf}_t = \ln(N/\text{df}_t)$

where

- tf_{dt} = [*# occurrences of t in d*],
- N = [*# documents in the corpus*],
- df_t = [*# documents in the corpus which contain t*],
- len_d = [*# words in d*].

The number of documents should be very similar to those listed above. Possible differences may depend on your regular expressions from assignment 1. Depending on exactly how you compute the similarity scores the **ordering of the documents can**

differ somewhat from those produced by your program – this is fine. You can debug your results by manually computing the tf_{dt} and len_d scores for the top ranked documents *d* for a term *t*. (The idf score does not influence the ranking since there is only one term.)

There will not be any examination of Task 2.1, it is merely a preparation for Task 2.2.

Task 2.2: Ranked Multiword Retrieval

Modify your program so that it can search for multiword queries, and present a list of ranked matching documents. All documents that include at least one of the search terms should appear in the list of search results.

When you have finished adding to the program, compile and run it, indexing the data set **`davisWiki`**. Select the "Ranked retrieval" option in the "Search Options" menu, and try the search queries:

zombie attack	**money transfer**
Found 249 matching document(s)	**Found 1598 matching document(s)**
0. JasonRifkind.f ... **1. Zombie_Walk.f ...** **2. Kearney_Hall.f ...** **3. Measure_Z.f ...** **4. Spirit_Halloween.f ...** **5. EmilyMaas.f ...** **6. AliciaEdelman.f ...** **7. TheWarrior.f ....** **8. Scream.f ...** **9. Zombies_Reclaim_the_Streets.f ...** *etc.*	**0. MattLM.f ...** **1. Angelique_Tarazi.f ...** **2. JordanJohnson.f ...** **3. Transfer_Student_Services.f ...** **4. NicoleBush.f ...** **5. Anthony_Swofford.f ...** **6. Title_Companies.f ...** **7. Transfer_Student_Association.f ...** **8. Munch_Money.f ...** **9. money.f ...** *etc.*

Our lists above were computed with the same length-normalized tf_idf scores for each query term as in Task 2.1, weighed together using cosine similarity.

At the review

To pass Task 2.2, you should be able to start the search engine and perform a search in ranked retrieval mode with a query specified by the teacher, that returns the correct number of documents in an order similar to the model solution used by the teachers. You should also be able to explain all parts of the code that you edited.

Task 2.3: Variants of cosine similarity and TF-IDF

This is a pen-and-paper task. Consider:

A. The query **`davis leadership`**
B. The contents of the file APILP.f
C. The contents of the file Davis_Funeral_Chapel.f

What is the term frequency of the words '**`davis`**' and '**`leadership`**'?

	`davis` ALILP.f	**`leadership`** ALILP.f	**`davis`** Davis_Funeral_Chapel.f	**`leadership`** Davis_Funeral_Chapel.f
tf	0	1	1	0

Now use your search engine (how?) to compute the idf of all terms present in both files (round to 4 decimal places). What are the **lengths** of these documents in the tf-space and in the tf×idf space (rounded to 4 decimal places)?

	`davis leadership`	ALILP.f	Davis_Funeral_Chapel.f
Euclidean, tf	1.4142	2.4495	2.2361
Manhattan, tf	2	6	5
Euclidean, tf × idf	3.8721	9.9688	12.132
Manhattan, tf × idf	4.3634	23.058	23.306

What is the **cosine similarity** between the query and the two documents (in the specified spaces using the specified length normalization, rounded to 4 decimal places)? Don't forget to use idf for the query terms in the two last rows.

	ALILP.f	Davis_Funeral_Chapel.f
Euclidean length, tf	0.2887	0.3162
Manhattan length, tf	0.0833	0.1
Euclidean length, tf × idf	0.3812	0.00592
Manhattan length, tf × idf	0.1463	0.00273

N = 17478 + 1 (query doc) I idf = ln (N / df) where df = retrieved document + 1 (presence in the query)
tf & tfidf vec query : (1, 1, 0, 0, 0, 0, 0, 0, 0, 0) & (0.5274, 3.836, 0, 0, 0, 0, 0, 0, 0, 0)
tf & tfidf vec ALILP : (0, 1, 1, 1, 1, 1, 1, 0, 0, 0) & (0, 3.836, 2.2149, 3.8312, 4.3007, 6.3348, 2.5404, 0, 0, 0)
tf & tfidf vec Funeral : (1, 0, 1, 0, 0, 0, 0, 1, 1, 1) & (0.5274, 0, 2.2149, 0, 0, 0, 0, 7.6893, 6.4015, 6.4729)

What is the cosine similarity (rounded to 4 decimal places) if the query coordinates are considered to be (1,1)?

	APILP.f	Davis_Funeral_Chapel.f
Euclidean length, tf	0.5773	0.6324
Manhattan length, tf	0.1667	0.2
Euclidean length, tf × idf	0.3884	0.3192
Manhattan length, tf × idf	0.1490	0.1474

I only imagine that we have the presence of davis & leadership without modifying old numbers (i.e idf)
Therefore we only replace the 0 in the vec by the corresponding value in the other vec

At the review

To pass Task 2.3, fill in the tables above, and be prepared to explain how you computed the numbers. Furthermore, you should be prepared to reason about the effects of different variations of cosine similarity and TF-IDF for the query **`davis leadership`**.

Task 2.4: What is a good search result?

We would like to assess whether ranked retrieval gives answers with higher precision and recall than the unranked intersection retrieval we tried in assignment 1.5.

In your search engine, select the option the "Ranked retrieval" option in the "Search options" menu. After having indexed the davisWiki dataset, give the same query as in Task 1.5:

graduate program mathematics

Inspect the **50 highest ranked** documents. If you already came across that document for the same query in Task 1.5, use the existing relevance label. Otherwise, assess the relevance of the document for the query. As in Task 1.5, use the following four-point scale:

(0) Irrelevant document. The document does not contain any information about the topic.
(1) Marginally relevant document. The document only points to the topic. It does not contain more or other information than the topic description.
(2) Fairly relevant document. The document contains more information than the topic description but the presentation is not exhaustive.
(3) Highly relevant document. The document discusses the themes of the topic exhaustively.

[E. Sormunen. Liberal relevance criteria of TREC—Counting on negligible documents? *ACM SIGIR*, 2002]

Write the results into a file using the following space-separated format, one line per assessed document:

`QUERY_ID DOC_ID RELEVANCE_SCORE`

where **`QUERY_ID = 1, DOC_ID`** = the name of the document, **`RELEVANCE_SCORE`** = [0, 1, 2, 3]. Like with Task 1.5, upload the file to Canvas under 'Assignment 2'.

It should again be noted that there is no objectively correct relevance label for a certain query-document combination! It is a matter of judgment. For difficult cases, write a short note on why you chose the label you did. At the review, you will present three difficult cases.

Plot a **precision-recall graph** for the returned top-50 list, and compute the **precision at 10**, **20**, **30**, **40**, and **50** (relevant documents = documents with relevance > 0).

Assume the total number of relevant documents in the corpus to be 100, and estimate the **recall at 10**, **20**, **30**, **40**, and **50**.

Compare the precision at 10, 20, 30, 40, 50 for ranked retrieval to the precision for unranked retrieval. *Which precision is the highest? Are there any trends?*

Do the same comparison of recalls. *Which recall is the highest? Is there any relation between precision at 10, 20, 30, 40, 50, and recall at 10, 20, 30, 40, 50?*

At the review

Less precision : lot of short irrelevant document ; Less recall : lot of document irrelevant (i.e. not all terms are present in the doc) ; Decreasing of precision for same recall : the 40th to 50th docs are irrelevant)

To pass Task 2.4, you should be able to show the text file with labeled documents, in the correct format. You should have emailed it before presenting. You should show the precision-recall graph for the 50 highest ranked documents, and be able to explain the concepts precision-recall graph and precision at K, and give account for these measures for the returned ranked top-50 document list.

Task 2.5: Computing PageRank with Power Iteration + combining PageRank with TF-IDF

The **`pagerank`** directory contains the file **`PageRank.java`**, which is compiled simply by

```
javac PageRank.java
```

The program is executed as follows:

`java -Xmx1g PageRank` *linkfile*

for instance

```
java -Xmx1g PageRank linksDavis.txt
```

The link file **`linksDavis.txt`** is also found in the folder **`pagerank`**. It contains the link structure of davisWiki. Each line has the following structure:

```
1;2,3,4,
```

meaning that webpage number **1** is linking to the articles in **2,3** and **4.** We are using numbers instead of the actual webpage names for the sake of brevity; however, you can translate from numbers to filenames by using the table in **davisTitles.txt**.

Note that the docIDs in **linksDavis.txt** and the internal docIDs that are produced on the fly in your index during indexing of the davisWiki corpus are NOT the same!

The first part of the task is to **extend the class PageRank.java so that it computes the pagerank of the davisWiki pages** given their link structure using the standard power iteration method.

You should be able to process the graph without using more than 1GB of heap space, and it should not take more than 1 minute or so to compute the pageranks. (If it takes more, you should be able to optimize the main loop).

Make sure your program prints the pagerank of the 30 highest ranked pages. Use the array **docName** to translate from internal ID numbers to the numbers in the **linksDavis.txt** file. Compare with the results in the file **davis_top_30.txt**.

Look up the titles of some documents with high rank, and some documents with low rank. Does the ranking make sense?

The second part of your task is to integrate the obtained pageranks for **linksDavis.txt** into the search engine we have been developing in Assignment 1 and Tasks 2.1-2.2. When doing a ranked query, make sure that the score is computed as a function of the tf-idf similarity score and the pagerank of each article in the result set. **Design the combined score function so that you can vary the relative effect of tf-idf and pagerank in the scoring.** You should pre-compute the pageranks and read them from file at the start of a search engine session.

You will need to add code to the **search** method, so that when this method is called with the **rankingType** parameter set to **RankingType.TF_IDF**, the system should perform ranked retrieval based on tf_idf score only, with the **rankingType** parameter set to **RankingType.PAGERANK**., only pagerank should be regarded, and with the **rankingType** parameter set to **RankingType.COMBINATION**, your combined score function is used to rank the documents.

When your implementation is ready, compile and run it, indexing the data set **davisWiki**. Select the "Ranked retrieval" option in the "Search Options" menu and the "Combination" option in the "Ranking Score" menu, and try the search queries listed in Task 2.2.

Each query should return the same number of matching documents as in Task 2.2. However, the ranking will vary depending on how you use the document pageranks in the score.

What is the effect of letting the tf_idf score dominate this ranking? What is the effect of letting the pagerank dominate? What would be a good strategy for selecting an "optimal" combination? (Remember the quality measures you studied in Task 2.3.)

Tf_idf : Straight to the point, no information, document with the word
PageRank : The most referenced document -> when we want to find information
Optimal Combination : use some queries ; compute precision & recall ; find the optimal values

At the review

To pass Task 2.5, you should show that the method returns a very similar top-30 ranking for **`linksDavis.txt`** to the one given. The difference in rank for a certain document should not be larger than ±2 positions, and the difference in pagerank value for the documents should not be larger than ±0.001. You should also be able to explain all parts of the code that you wrote.

Furthermore, you should present a function for combining tf-idf and pagerank scores where the influence of the two factors can be varied.

You should be able to start the search engine and perform a search in combination, ranked retrieval mode with a query specified by the teacher, that returns the correct number of documents, and be able to discuss the effect of tf-idf and pagerank on the subsequent ranking.

You should also be able to explain all parts of the code that you edited and be able to discuss the question in italics above.

Task 2.6: Cosine similarity with Euclidean length (C)

In task 2.1, the tf_idf score for a document *d* and a query term *t* was computed using the following formulas:

- $\text{tf\_idf}_{dt} = \text{tf}_{dt} \times \text{idf}_t / \text{len}_d$
- $\text{idf}_t = \ln(N/\text{df}_t)$

where

- tf_{dt} = [*# occurrences of t in d*],
- N = [*# documents in the corpus*],
- df_t = [*# documents in the corpus which contain t*],
- len_d = [*# words in d*].

This computation provides an approximation of the cosine similarity between the document and the query. In this task, you are going to assess how good this approximation is.

In the first part of the task, you need to add code to the **`Indexer`** and **`Engine`** classes to compute Euclidean lengths of every document in the corpus and store them in the file on the disk. This computation should be performed **only once** and all the subsequent restarts should load the Euclidean lengths from the file (if it exists). Note that the Euclidean length should be computed using **all terms present in the document**!

In the second part of the task, you need to implement ranked retrieval with tf_idf scores **normalized by the Euclidean length of document *d***. That is, tf_idf is computed as above, with the exception that len_d now is [*Euclidean length of d*].

Here are the results for the same queries as in Task 2.2 using the new version of tf_idf scores:

zombie attack	money transfer
Found 249 matching document(s)	**Found 1598 matching document(s)**
0. Zombie_Walk.f ... **1. JasonRifkind.f ...** **2. Measure_Z.f ...** **3. Zombie_Attack_Response_Guide.f ...** **4. Kearney_Hall.f ...** **5. Spirit_Halloween.f ...** **6. Zombies_Reclaim_the_Streets.f ...** **7. Scream.f ...** **8. Furly707.f ...** **9. Biological_Disasters.f ...** *etc.*	**0. Transfer_Student_Services.f ...** **1. MattLM.f ...** **2. Transfer_Students.f ...** **3. JordanJohnson.f ...** **4. Angelique_Tarazi.f ...** **5. money.f ...** **6. Joanna_Villegas.f ...** **7. Munch_Money.f ...** **8. ScarlettYing.f ...** **9. Jeserah.f ...** *etc.*

Compare the results for the queries in task 2.2 and compute precision@10 and recall@10 for both cases, i.e. (1) when normalizing with the number of words and (2) when normalizing with the Euclidean distance of the document. *Which of them is better and why?*

Zombie : Manhattan : precision 6/10 ; recall 6/100
Zombie : Euclidean : precision 6/10 ; recall 6/100
Money : Manhattan : precision 1/10 ; recall 1/100
Money : Manhattan : precision 1/10 ; recall 1/100
Found the same value but euclidean seems to put more value on big files

At the review

To pass Task 2.6, you should be able to start the search engine and perform a search in ranked retrieval mode with a query specified by the teacher. The correct number of documents should be returned, in an order similar to the model solution used by the teachers. You should also be able to explain all parts of the code that you edited and answer the question in italics.

Task 2.7: Monte-Carlo PageRank Approximation (B)

The task is now to implement the Monte-Carlo methods 1,2,4 and 5 for approximate PageRank computation mentioned in Lecture 5 and in the paper by Avrachenkov et al. listed as course literature.

Run these four variants on **`linksDavis.txt`**, using c = 0.85 and several different settings of N (the number of initiated walks). Compare the four method variants and settings of N in terms of how fast they converge and how similar the solution is to the exact solution. Implement the following goodness measure:

> The sum of squared differences between the exact page ranks and the MC-estimated page ranks for the **30 documents with the highest exact page rank** in **`linksDavis.txt`**.

Plot this goodness measure for all four methods as a function of N.

What do you see? Why do you get this result? Explain and relate to the properties of the (probabilistic) Monte-Carlo methods in contrast to the (deterministic) power iteration method.

Convergence time : 4 ; 5 ; 2 ; 1 (fastest to lowest)

Do your findings about the difference between the four method variants and the dependence of N support the claims made in the paper by Avrachenkov et al.?

Finally, use your favorite Monte-Carlo method to approximate the page ranks of the full Swedish Wikipedia link structure (in the file **`linksSvwiki.txt`**). Iterate until the top 30 documents are stable.

At the review

To pass Task 2.7, you should show a record of your experimentation with the four method variants and their *N* parameter settings for the **`linksDavis.txt`** graph.

You should be able to discuss the questions in italics, be able to discuss the differences between the four variants, and explain all parts of the code that you wrote.

Finally, show your list of 30 top documents for the **`linksSvwiki.txt`** graph. Argue why they are correct by looking up titles of top documents in the file **`svwikiTitles.txt.`**

Task 2.8: Hubs and Authorities (A)

In this task you will implement another algorithm that rates (web) documents by analyzing the links between them. The algorithm is called HITS (Hyperlink-Induced Topic Search) also known as Hubs and Authorities (please refer to section 21.3 of the textbook for more details).

HITS was proposed at about the same time as PageRank, but is less popular. In this task, you'll become an unprejudiced experimental researcher comparing two algorithms and investigating their strengths and weaknesses.

In the skeleton for assignment 2, there is a class called **`HITSRanker.java`**, which should be put into the **`ir`** folder.

Your task is to **extend the class `HITSRanker.java` so that it computes the hub and authority scores for the davisWiki pages** given their link structure. As in task 2.5, you can find the link structure of davisWiki in the file **`linksDavis.txt`** and the titles of articles in **`davisTitles.txt`**, respectively.

Start by implementing **`readDocs`** and **`iterate`** functions of the **`HITSRanker`** class. Re-use as much code as possible from the PageRank skeletons, provided for task 2.5. Declare the convergence of the HITS algorithm if both hub and authority scores do not change more than a predefined value **`EPSILON`** from iteration to iteration.

To make debugging easier for you, there are lists of top 30 hub and authority scores for the whole davisWiki corpus in the files **`davis_hubs_top_30.txt`** and **`davis_authorities_top_30.txt`** respectively. You can test your implementation by running **`HITSRanker`** as follows:

```
javac -cp . -d classes ir/HITSRanker.java

java -cp classes ir.HITSRanker linksDavis.txt davisTitles.txt
```

This should create two files: **`hubs_top_30.txt`** and **`authorities_top_30.txt`**.

NOTE: that you should change the paths to **`linksDavis.txt`** and **`davisTitles.txt`** to point to the actual files on your machine.

Compare your 30 highest-ranked hub and authority scores to the ones in the provided files. *Does the ranking make sense? How does it compare to page ranks?*

After you have implemented the HITS algorithm, you'll need to integrate HITS ranking into your search engine. On the way, there are several issues to be addressed:

1. Unlike PageRank, the HITS method should be run on the fly and only on the query-specific subset of documents. How should one select this subset of documents correctly?
2. The HITS algorithm provides two scores for each document in the subset, but the search engine can show only one score. How should one combine these two scores in a meaningful way? Can we use a linear combination?

After you have addressed the issues listed above, you'll need to integrate HITS ranking into your search engine GUI, making it possible to choose **`HITS`** in the "Ranking Score" menu and have the query results ordered accordingly.

At the review

To pass Task 2.8, you should show that your implementation returns a very similar top-30 hub and authorities scores for **`linksDavis.txt`** to the ones given. You should explain how you have addressed the HITS integration issues listed above and demonstrate that using HITS ranking with your search engine is possible. You should be able to list similarities and differences between PageRank and HITS and argue about the advantages and disadvantages of each algorithm.

Similarities:
Link Analysis: Both algorithms use link analysis as the core part of their approach, where the structure of the web is analyzed to determine the quality and relevance of web pages.
Iterative Algorithms: PageRank and HITS are iterative, meaning they refine their results through multiple rounds of calculations to converge on a stable set of scores for web pages.
Authority based: Each algorithm assigns scores that aim to measure the authority or importance of web pages based on the idea that links from other pages are akin to votes or endorsements.
Differences:
Approach to Ranking:

PageRank evaluates the importance of a page based on the number and quality of links to it. It operates under the assumption that more important websites are likely to receive more links from other websites.
HITS identifies two distinct scores for each page: Authority, which measures the value of the content of the page, and Hub, which measures the quality of its links to other pages. It assumes that good hubs link to good authorities and vice versa.
Computation Complexity:

PageRank requires less computational effort compared to HITS because it calculates a single score (PageRank score) for each page, which is independent of a query.
HITS initially seems less computationally intensive because it operates on a smaller, query-dependent subgraph of the web. However, it must be recalculated from scratch for each new query, which can make it more resource-intensive in practice.
Query Dependence:

PageRank is query-independent; it pre-computes a global ranking of web pages that is used for any query.
HITS is query-dependent; it calculates hub and authority scores based on the specific set of pages relevant to a given search query.
Advantages:
PageRank:
Scalability: It's more scalable because it calculates a global score for each page that can be reused across different queries.
Simplicity: The concept is straightforward, making it easier to implement and understand.
HITS:
Relevance: By recalculating for each query, it can potentially provide more relevant results for specific searches.
Distinction between hubs and authorities: This can provide more nuanced insights into the nature of web pages and their roles in the information ecosystem.
Disadvantages:
PageRank:
Manipulation: It can be more susceptible to manipulation through link farms and other SEO techniques.
Outdated Information: Since it's computed less frequently, it might not reflect recent changes to the web as quickly.
HITS:
Computational Resources: Requires more computational resources for large datasets because it recalculates for each query.
Initial Set Selection: The effectiveness of HITS can be sensitive to the choice of the initial set of pages for a given query.