Wright State University

Department of Computer Science and Engineering

CS7800 Spring 2024

T. K. Prasad

Assignment 1 (Due: February 15) (10 pts)

Objective

This assignment offers an exciting opportunity to explore the realm of information retrieval systems. The main goal of this assignment is to introduce you to essential concepts like term generation, indexing, and query processing algorithms. You will apply these concepts to develop a basic information retrieval system during this project. This project offers you the opportunity to design and implement your own information retrieval system in Python using scikit-learn APIs, and it is structured into two distinct phases:

- Phase I: Building the indexing and search application.
- Phase II: Assessing the performance of your search engine.

You will be tasked with building your search engine in Python and evaluating it using the Cranfield Dataset. You can choose to work individually or in a team of up to three individuals. The Cranfield Dataset contains 225 queries (search terms), 1400 documents, and 1836 evaluations derived from 225 selected queries. When extracting data from this dataset, ensure that you exclusively use the body ("(.W)") for both documents (cran.all) and queries (query.txt). The Cranfield Dataset organizes documents and queries into different distinct parts or tags, including (.T) for title, (.A) for author, (.W) for body, (.I) for ID, among others.

Here is a brief description of the dataset components:

- **cran.all:** The Cranfield collection corpus file containing various fields, with only the (.W) field/tag being usable in this assignment.
- **query.txt:** A collection of 225 queries that can be utilized to assess the retrieval performance of your system, with the (.W) field/tag being the relevant one for this assignment.
- **qrels.txt:** Query relevance judgments for each query, with each query accompanied by a set of relevant documents listed in separate lines.
- **README.txt**: A detailed explanation of each column in the dataset files is provided in this file.

Phase I: Indexer and Query Processor

In Phase I, we will leverage the Scikit-learn library, a powerful open-source machine learning tool for Python, to create an efficient Indexer and Query Processor. Our primary objective is to process

the Cranfield Dataset and provide answers to standard queries. Below, we outline the key modules required for Phase I:

Vectorization and Indexer Module:

The Indexer module is integral in transforming the Cranfield dataset's documents into a searchable index/matrix. Considering the Cranfield collection's unique format — storing documents in a single file — it is imperative to first separate these documents into multiple files for effective indexing. This process is facilitated by employing vectorization techniques using scikit-learn. The following steps outline the critical components of the Indexer module:

1. **Data Preparation:** Your initial task is to prepare two sets: a corpus of documents (the training set) and a set of queries (the test data). To accomplish this, you will leverage two distinct vector space models by employing different vectorization techniques within the scikit-learn framework.
    - **NOTE:** Understanding the distinctions between `fit()`, `transform()`, and `fit_transform()` is crucial If you are unfamiliar with the differences between fit(), transform(), and fit_transform(), The appropriate application of these functions is vital for accurate indexing and query processing. It's recommended to explore additional resources, including documentation, to gain a comprehensive understanding.
2. **Text Preprocessing:** Text preprocessing is a fundamental step in Natural Language Processing (NLP) systems. Its impact on system performance is significant. To ensure the efficiency and effectiveness of your system, adhere to industry-standard text processing techniques, including:
    - **Tokenization:** Divide a character sequence (text document) into meaningful units called tokens, while removing irrelevant characters like punctuation.
    - **Case-Folding:** Normalize text documents by converting all letters to lowercase. This approach ensures that instances of words with varying capitalization are treated consistently.
    - **Stop-Word Removal:** Eliminate common words, known as stop-words, from the text. It is required to use `sklearn.feature_extraction.text.ENGLISH_STOP_WORDS` for your list of stop-words.

In this section of your assignment, you'll face a dual challenge. First, you need to proficiently utilize the default Binary Vectorizer. Second, your task involves replacing the standard TF-IDF calculation in the library with a custom implementation that creatively modifies the traditional IDF calculation. For both of these vectorizers, it's essential to maintain consistency and accuracy in your data handling and analysis by adhering to text preprocessing standards. The following outline offers precise instructions for your implementation:

Part 1: Binary Vectorizer

- **Usage**: Employ the Binary Vectorizer, which assigns '1' if a term is present in a document/query and '0' otherwise.
- **Configuration**: Ensure to enable `stop_words` and `lowercase` options. Tokenization is handled by default.
- **Reference**: Consult the scikit-learn documentation for more details and implementation guidelines.

Part 2: Custom TF-IDF Vectorizer

- **Background**: TF-IDF is a key measure in text mining, indicating the significance of a word in a document within a corpus. Traditionally, the IDF component uses a logarithmic scale. However, you will implement a variant where the IDF is the reciprocal of the document frequency.
- **Usage**: Implement a Custom TF-IDF Vectorizer that interfaces seamlessly with `CountVectorizer`. Your vectorizer should compute term frequencies as usual but calculate the Inverse Document Frequency (IDF) as the reciprocal of the document frequency (without using the logarithmic scale).
- **Configuration**:
    1. **Term Frequency (TF)**: Utilize the standard `CountVectorizer` for the initial calculation of term frequencies. This involves counting the occurrences of each term in each document.
    2. **Custom IDF Calculation**: Modify the IDF component to be the simple reciprocal of the document frequency. This means for each term, calculate IDF as 1 divided by the number of documents containing that term.
    3. **Integration**: Ensure your custom TF-IDF logic is embedded within a transformer that can be used in conjunction with `CountVectorizer`. This transformer should redefine the `fit_transform` method to automatically apply your custom IDF calculation.
    4. **Additional Features**: While customizing the IDF calculation, you are required to leverage `CountVectorizer`'s functionalities like `stop_words` & `lower_case` removal.
- **Implementation Hint**: Create a custom transformer class that inherits from scikit-learn's base classes. This class should override the necessary methods to alter the standard TF-IDF calculation with your custom IDF logic. Remember to ensure compatibility with the scikit-learn pipeline, allowing for smooth integration and usage. An illustrative screenshot is provided for further guidance.

```
# Sample documents
documents = [
    "the quick brown fox jumps over the lazy dog",
    "the quick brown fox",
    "lazy dogs run fast",
    "quick brown foxes leap over lazy dogs in the park"
]

# Convert documents into a matrix of token counts
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(documents)

# Apply the custom TF-IDF transformation
transformer = CustomTfidfTransformer()
tfidf_matrix = transformer.fit_transform(X)
```

The Querying processor:
All queries will undergo the same series of preprocessing operations that were previously employed during the document indexing process. After this preprocessing step, we will assess the similarity between queries and documents using two distinct similarity metrics within the two vector space models. The selected similarity metrics are Cosine Similarity and Manhattan Distance. It is imperative to exclusively employ the following similarity measures for this task:

- sklearn.metrics.pairwise.cosine_similarity
- sklearn.metrics.pairwise.manhattan_distances

After calculating the similarity, each element in the result matrix represents similarity between one query and one document. Do not normalize document vector or query vectors to a unit length.

Similarity Descriptions:

- The cosine similarity can be calculated for a document and a query represented by their TF-IDF vectors d_i and q_j as follows:

$$\cos(d_i, d_j) = \frac{d_i \bullet q_j}{\|d_i\| \|q_j\|} = \frac{\sum_{k=1}^{n} d_{i_k} q_{j_k}}{\sqrt{\sum_{k=1}^{n} d_{i_k}^2} \sqrt{\sum_{k=1}^{n} q_{j_k}^2}}$$

  where n is the total number of terms, and k represents the term being iterated over. The cosine similarity is 0 for orthogonal vectors, and 1 for identical vectors. Again, we are not asking you to manually code any of the two-similarity equations.

- The Manhattan distance, also known as the L1 norm or "city block" distance, measures the distance between two points in a space by summing the absolute differences of their coordinates. For a document and a query represented by their TF-IDF vectors d_i and q_j the Manhattan distance can be calculated as follows:

$$\text{Manhattan}(d_i, q_j) = \sum_{k=1}^{n} \left| d_{i_k} - q_{j_k} \right|$$

  Here, n is the total number of terms, and kk represents the term being iterated over. In this formula:
  - d_{i_k} is the TF-IDF weight of term k in document i.
  - q_{j_k} is the TF-IDF weight of term k in query j.
  - The symbol | . | denotes the absolute value.

  The Manhattan distance effectively adds up the absolute differences between the corresponding elements of the two vectors. In the context of text analysis, this measure calculates how far apart two documents, or a document and a query are, considering each term's weight. The smaller the Manhattan distance, the more similar the documents are.

  It's important to note that unlike cosine similarity, which ranges from -1 to 1, Manhattan distance is not bounded and depends on the magnitude of differences in the term weights. In simple terms, it's like measuring the distance by walking along city blocks, moving either horizontally or vertically, but not diagonally, to transition from one point to another.

Phase II: Evaluation

- Get the indices (indexes) of the 10 most relevant documents for each query (225) for both vector space models (TFIDF and Binary) and both distance measures (Manhattan distance and Cosine Similarity). You should end up with **four** lists, consisting of 225 queries each, where each query item contains the 10 most relevant documents for each model-measure configuration. Note: utilizing a list of lists, dictionary of lists, or NumPy array, might be a good choice. The final shape of each of the data structures should be (225,10).
- Calculate the precision, recall and F-score from the lists of query relevant documents, retrieved in the previous step, against the list of relevant documents in qrels.text.
- Discuss how precision, recall, and F-score may vary or may not vary between different vector models and between different similarity measures used in this assignment.
- Discuss the effectiveness/ineffectualness of the two similarity measures and the two vector models used at retrieving the correct document.
- Discuss any other relevant insights about this assignment.
- Graphically display the Precision, Recall, and F-scores for the two similarity measures and two vector models using matplotlib.pyplot.bar. Your task is to generate a total of 12 graphs, each saved as .png files to your working directory. Do not display these graphs while your application is running; they should be saved in the background. Figure 1 demonstrates exactly how your graph should look.
- Finally, display in the terminal the mean and maximum Precision, Recall, and F-scores for the two similarity measures and the two vector models. Figure 2 demonstrates **exactly** how your terminal output should look. I repeat, do not deviate from this format at all as we will be using a grading script.

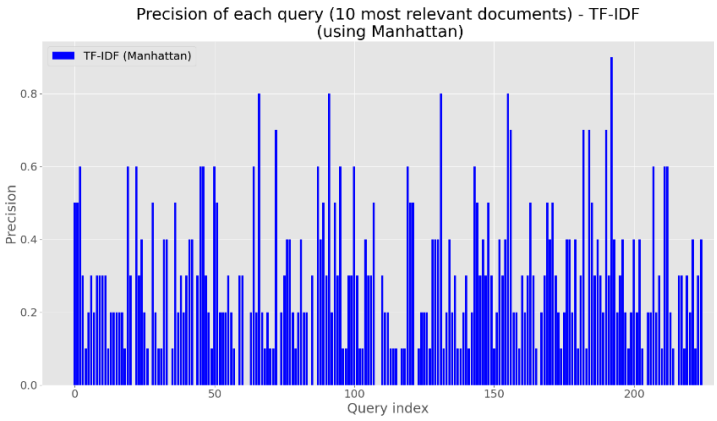

Figure 1: example of how you should graphically display your precision, recall, and f-score.

```
{
    'Binary': {'f': {'cos': (mean, max),
                     'man': (mean, max)},
               'p': {'cos': (mean, max),
                     'man': (mean, max)},
               'r': {'cos': (mean, max),
                     'man': (mean, max)}},
    'TFIDF': {'f': {'cos': (mean, max),
                    'man': (mean, max)},
              'p': {'cos': (mean, max),
                    'man': (mean, max)},
              'r': {'cos': (mean, max),
                    'man': (mean, max)}}
}
```

Figure 2: example of your exact format for the terminal output

Setup:

- Download the Cranfield dataset. It contains three files: cran.all - the document collection, query.text - the sample queries, qrels.text - the relevant query-document pairs. Check the README.txt for more details.

- Use Python 3 or higher
  - Do not use Jupiter notebook
- You may need NumPy.
  - `import numpy as np`
- Install scikit-learn:
  - See: https://www.activestate.com/resources/quick-reads/how-to-install-scikit-learn/
- Name your well-documented Python script as *assignment1.py*.
- Students are required to use a Python virtual environment. For those unfamiliar with Python virtual environments here are two reference articles explaining how to use and activate:
  - See: https://uoa-eresearch.github.io/eresearch-cookbook/recipe/2014/11/26/python-virtual-env/
  - See: https://realpython.com/lessons/creating-virtual-environment/
- Once you finish this assignment and are ready to submit it to pilot, you need to create a *requirements.txt* file with all the libraries used in your development. Use the following command only after you've activated your Python environment. This is important to make sure that we can run your code.
  - `source venv/bin/activate`
  - `pip list --format=freeze > requirements.txt`
  - `deactivate`
  - `See:`https://openclassrooms.com/en/courses/6900846-set-up-a-python-environment/6990546-manage-virtual-environments-using-requirements- files
- Important note: do not use any other nonstandard Python library's.

Debug:

Please test/debug all your programs thoroughly. By designing tests, ask yourself questions like:

- Are the stop-words really removed?
- What is the number of terms in the dictionary? Do they make sense?
- Does your document-term matrix have the right shape?
- How do you confirm that TF-IDF values are computed correctly?
- Do you also convert queries to terms?
- How do you confirm similarity is computed correctly?
- How do you confirm Cosine Similarity is computed correctly?
- Are your selected sample queries getting the same results as you expect for Boolean model processing?
- Are your selected sample queries getting the same results as you expect for vector model processing?

These are just a few questions I would suggest you ask yourself before submitting your assignment. I would expect you to perform more testing than what is suggested above.

Deliverables

TURN IN: Upload one tar archive file or Zip file per team that contains the following files:

- **Code and accompanying documentation***:* Include well-documented source code for the entire project.
- **Evaluation information***:* Discuss, in a PDF file, the evaluation metrics to be used to quantify the quality of the search results, and determine the quality of results (e.g., precision, recall, f-score) for all the top 10 relevant document query results from the Cranfield collection and report the evaluation metrics for the entire test set. This document should also contain the screenshots of your graphs and terminal output. Additionally, all team member names, UIDs, and email addresses must be included in this document.
- **README.txt:** This document should briefly explain your application, how to launch the application, any external libraries used, what version **of Python 3 used**, all team members names and UIDs, and any other relevant information. The more information you provide in this document the easier it is for us to grade andgive partial credit if something does not work on our system.
- **Application execution**: Ensure that your Python program is executable from the command-line. To execute your code, use the following command:
  - *python3 assignment1.py*
  - It is essential that your assignment1.py is self-contained, capable of running on any computer, and can be executed from the command line using the exact syntax provided above.
  - Your application should automatically save the 12 graphs generated to the working directory and should not display them during execution.
  - Avoid using absolute paths for input files or any data files; all file paths should be relative to your working directory.
  - We recommend that each team member actively participates in the development process and separately tests the application on multiple installations or computers to ensure compatibility and to avoid hardcoding any specific configurations.
  - Please note that using Jupyter Notebook is not allowed, as your application must launch from the command-line using the specified syntax.
  - Any deviations from these guidelines will result in a 30% penalty.
- **Input and output files**: All file(s) must be placed in your working directory, all generated output file(s) must also be placed inyour working directory, and no subdirectories.
- **requirements.txt:** All the libraries used in your virtual Python environment must be stated.
- **Upload** the archive *asg1.zip* onto *Pilot $>$ Dropbox $>$ Assignment 1*folder by **February 15**. (Only one submission per team.) Any member of the team can be required to demonstrate your program to us (if necessary) and be prepared to answer questions about its design, implementation, and comparative evaluation.
- **Environment setup requirements**: Any deviation from what was discussed in this section will result in a 30% reduction for each deviation in your total grade. Therefore,

I strongly suggest you make sure you follow this document closely and ask questions on *discord* for clarification.

Grading Criteria

You must obtain a PASS on this assignment to PASS the course. At the minimum, your code should compile, process some queries, and return reasonable results. A passing grade is 60%.

Assignments are designed to help you learn the core concepts and are the primary course "homework". Corrupt files or other computer problems will not be considered a valid excuse to extend the deadline. It is your responsibility to regularly back-up your work. We strongly suggest that you save your work to multiple locations to aid in the recovery of corrupt files. If you have questions regarding the project, we (the GTA, the grader and the instructor) are there to help.

Assignments that are submitted late will incur a penalty of 25% reduction on total grade per day the assignment is late. The project must be turned in on Pilot as described in the project description to receive full credit. Assignments emailed to the GTA, or professor will receive an immediate 25% reduction in total grade because the whole purpose of Pilot is to streamline communication.

Cheating:

Please do not copy other team's work, do not copy other projects found on the Internet, do not blindly use AI assistance (ChatGPT/Copilot), or use any outside resource without proper citation. In short, plagiarism will get you a zero on this assignment and potentially an F grade in the class. We are not obsessed with looking for cheating, but if we see something suspicious, we will investigate and then refer it to the Office of Judicial Affairs. This is more work for us and is embarrassing for everyone. Again, please don't; this has been a problem in the past. If the rules are unclear or you are unsure of how they apply, ask the instructor, GTA, or grader beforehand. The academic integrity policy as available online.

No deadline extension will be given.