COMP24011 Lab 4:
BM25 for Retrieval-Augmented Question Answering

Riza Batista-Navarro and Francisco Lobo
Academic session: 2023-24

Introduction

In this exercise, you will develop your own implementation of the BM25 scoring algorithm, one of the most popular methods for information retrieval. Apart from the traditional uses of information retrieval methods in the context of search engines and document ranking, they have recently been employed to enhance the question answering (QA) capabilities of generative large language models (LLMs). Such models such as ChatGPT, can answer questions based on knowledge learned during their training on large amounts of textual data. However, they suffer from well-known limitations, including their tendency to hallucinate (i.e., make up answers that are factually wrong), as well as biases that they learned from the training data.

A workaround to these issues is the integration of an information retrieval module into the question answering pipeline, in order to enable the LLM to access factual information stored in relevant documents that can be used by the model in producing its output.

If you follow this manual all the way to the end, you will have the opportunity to observe how BM25 enables an LLM to provide more accurate answers to questions. Your main task for this exercise, however, is to implement pre-processing techniques, compute the BM25 score of each (pre-processed) document in relation to a (pre-processed) question, and return the topmost relevant documents based on the scores.

For this exercise, you are provided with the following text files as resources:

`transport_inventions.txt`	The content of this file was drawn from Wikipedia's timeline of transportation technology. We will consider this file as a *corpus*, i.e., a collection of documents, whereby each line corresponds to one *document*. Given that there are 10 lines in the file, this corpus consists of 10 documents.
`music_inventions.txt`	The content of this file was drawn from Wikipedia's timeline of music technology. We will consider this file as another corpus. As in the first corpus, each line corresponds to one document. Given that there are 10 lines in the file, this corpus consists of 10 documents.
`stopwords_en.txt`	This file contains a *stop word list* taken from the Natural Language Tooklkit (NLTK). This is a list of words that are commonly used in the English language and yet do not bear meaning on their own. Every line in the file is a stop word.

If you make changes to the contents of these files, this will change the expected behaviour of the lab code that you're developing, and you won't be able compare its results to the examples in this manual. But you can always use `git` to revert these resources to their original state ☺

To complete this lab you will need a third-party stemming tool called PyStemmer. You can install it by issuing the following command

```
$ pip install pystemmer
```

The BM25 Retrieval System

Once you refresh the `lab4` branch of your GitLab repo you will find the following Python files.

`run_BM25.py`	This is the command-line tool that runs each separate NLP task according to the subcommand (and the parameters) provided by the user. It contains the `RunNLP` class.
`nlp_tasks_base.py`	This module contains the `NLPTasksBase` "abstract" class that specifies the signatures of four methods you need to implement, and implements the interface used in `RunNLP`.
`nlp_tasks.py`	This is the module that you need to complete for this exercise. It contains the `NLPTasks` class that is derived from `NLPTasksBase`, and must implement its abstract methods in order to complete the BM25-based retrieval of documents relevant to a given question.

In order to successfully complete this lab you will need to understand both `nlp_tasks_base.py` and `nlp_tasks.py` but you do not need to know the details of how `run_BM25.py` is coded.

Once you complete this exercise, the BM25 tool will be able to obtain the documents most relevant to a given question. This BM25 retrieval system provides comprehensive help messages. To get started run the command

```
$ ./run_BM25.py -h
usage: run_BM25.py [-h] -c CORPUS [-w STOPWORDS] [-s]
                   {preprocess_question,preprocess_corpus,IDF,BM25_score,top_matches}
                   ...

options:
  -h, --help            show this help message and exit
  -c CORPUS, --corpus CORPUS
                        path to corpus text file (option required except for
                        the preprocess_question command)
  -w STOPWORDS, --stopwords STOPWORDS
                        path to stopwords text file (option required unless
                        stopwords are located at ./stopwords_en.txt)
  -s, --stemming        enable stemming

subcommands:
  select which NLP command to run

  {preprocess_question,preprocess_corpus,IDF,BM25_score,top_matches}
    preprocess_question
                        get preprocessed question
    preprocess_corpus   get preprocessed corpus
    IDF                 calculate IDF for term in corpus
    BM25_score          calculate BM25 score for question in corpus document
    top_matches         find top scoring documents in corpus for question
```

Notice that for most subcommands you need to specify which corpus to work with, as you'll have the 2 choices described in the Introduction: `transport_inventions.txt` or `music_inventions.txt`. On the other hand, unless you move the stopwords list to another directory, you should not need to give its location.

The tool has a boolean flag that controls if stemming should be applied when pre-processing text. By default it is set to `False`, but you can set it to `True` using the stemming option. This will affect the way your text preprocessing code for Task 1 below should work.

The BM25 tool supports five subcommands: `preprocess_question`, `preprocess_corpus`, `IDF`, `BM25_score` and `top_matches`. The first two will call your text pre-processing implementation, the others will call the corresponding functions that you'll develop in Tasks 2 to 4 below. Each of these subcommands has its own help message which you can access with commands like

```
$ ./run_BM25.py top_matches -h
usage: run_BM25.py top_matches [-h] question n

positional arguments:
  question    question string
  n           number of documents to find

options:
  -h, --help  show this help message and exit
```

The BM25 tool will load the stopwords list and corpus as required for the task. For example, running the command

```
$ ./run_BM25.py preprocess_question "Who flew the first motor-driven airplane?"
nlp params: (None, './stopwords_en.txt', False)
debug run: preprocess_question('Who flew the first motor-driven airplane?',)
ret value: flew first motor driven airplane
ret count: 32
```

will not load the corpus as text pre-processing is only applied to the given question string. Note that text pre-processing should, in general, return a different value if stemming is enabled. In fact, for the same question of the previous example you can expect

```
$ ./run_BM25.py -s preprocess_question "Who flew the first motor-driven airplane?"
nlp params: (None, './stopwords_en.txt', True)
debug run: preprocess_question('Who flew the first motor-driven airplane?',)
ret value: flew first motor driven airplan
ret count: 31
```

To pre-process the text of a whole corpus you should use the `preprocess_corpus` subcommand. For example, once you've finished Task 1 you should get

```
$ ./run_BM25.py -s -c music_inventions.txt preprocess_corpus
nlp params: ('music_inventions.txt', './stopwords_en.txt', True)
debug run: preprocess_corpus()
ret value: [
  '1940 karl wagner earli develop voic synthes precursor vocod',
  '1941 commerci fm broadcast begin us',
  '1948 bell laboratori reveal first transistor',
  '1958 first commerci stereo disk record produc audio fidel',
  '1959 wurlitz manufactur sideman first commerci electro mechan drum machin',
  '1963 phillip introduc compact cassett tape format',
  '1968 king tubbi pioneer dub music earli form popular electron music',
  '1982 soni philip introduc compact disc',
  '1983 introduct midi unveil roland ikutaro kakehashi sequenti circuit dave smith',
  '1986 first digit consol appear']
ret count: 10
```

Assignment

For this lab exercise, the only Python file that you need to modify is `nlp_tasks.py`. You will develop your own version of this script, henceforth referred to as "your solution" in this document.

Before you get started with developing this script, it might be useful for you to familiarise yourself with how the `NLPTasksBase` "abstract" class will initialise your `NLPTasks` objects:

- The documents in the specified corpus are loaded onto a list of strings; this list becomes the value of the field `self.original_corpus`
- The stop words in the specified stop word list file are loaded onto a list of strings, which becomes the value of the field `self.stopwords_list`
- If stemming is enabled, an instance of the third-party `Stemmer` class is created and assigned to the field `self.stemmer`

In addition, the pre-processing of the corpus and of the question strings is done automatically in the `NLPTasksBase` abstract class. The pre-processed text for these become available as the fields `self.preprocessed_corpus` and `self.preprocessed_question`, respectively.

Task 1: In your solution, write a function called `preprocess` that takes as input a list of strings and applies a number of pre-processing techniques on each of the strings. The function should return a list of already pre-processed strings.

Pre-processing involves the following steps, in the order given:

1. removal of any trailing whitespace
2. lowercasing of all characters
3. removal of all punctuation
4. removal of any stop words in the list contained in the specified stop word list
5. stemming of all remaining words in the string if stemming is enabled.

In relation punctuation removal, it is important to note the following:

- For a standard definition of what counts as a punctuation, you can use the values returned by the `string.punctuation` constant in Python.
- Avoid merging any tokens unnecessarily. For instance, in the above examples, the removal of the hyphen in *"motor-driven"* and the single quote in *"world's"* was done in such a way that the separation of corresponding tokens was preserved, leading to e.g., *"motor" "driven"* (instead of *"motordriven"*) and *"world" "s"* (instead of *"worlds"*).

As for applying the third-party stemming tool, please refer to PyStemmer's documentation, to find how one can call the `stemWords` function of a `Stemmer` object.

You can verify that your function behaves correctly on the command line. In addition to the examples in the previous section, note that in some cases stemming will not change the pre-processed result. For example, you should obtain the following output:

```
$ ./run_BM25.py preprocess_question \
                "When did the world's first underground railway open?"
nlp params: (None, './stopwords_en.txt', False)
debug run: preprocess_question("When did the world's first underground railway open?",)
ret value: world first underground railway open
ret count: 36
$ ./run_BM25.py -s preprocess_question \
                "When did the world's first underground railway open?"
nlp params: (None, './stopwords_en.txt', True)
debug run: preprocess_question("When did the world's first underground railway open?",)
ret value: world first underground railway open
ret count: 36
```

Task 2: In your solution, write a function called `calc_IDF` that calculates the inverse document frequency (IDF) of a given term (i.e., a token or word) in a pre-processed corpus. The score should be returned as as a float.

Since IDF is calculated based on a pre-processed corpus, this function will always be called after the `preprocess` function (Task 1) has been applied to the corpus. As explained, the result of this can be accessed as the field `self.preprocessed_corpus`.

You can verify that your function behaves correctly on the command line. For example, you should obtain the following output:

```
$ ./run_BM25.py -s -c transport_inventions.txt IDF airplan
nlp params: ('transport_inventions.txt', './stopwords_en.txt', True)
debug run: IDF('airplan',)
ret value: 0.8016323462331664
$ ./run_BM25.py -c transport_inventions.txt IDF first
nlp params: ('transport_inventions.txt', './stopwords_en.txt', True)
debug run: IDF('first',)
ret value: -0.531478917042255
```

Task 3: In your solution, write a function called `calc_BM25_score` that calculates the BM25 score for a pre-processed question (a string) and a pre-processed document that is specified by its index in the corpus (an integer, starting from zero). The score should be returned as a float.

As explained above, the pre-processed question and corpus can be accessed as the fields `self.preprocessed_question` and `self.preprocessed_corpus`, respectively.

You can verify that your function behaves correctly on the command line. For example, you should obtain the following output:

```
$ ./run_BM25.py -s -c transport_inventions.txt BM25_score \
                "flew first motor driven airplan" 4
nlp params: ('transport_inventions.txt', './stopwords_en.txt', True)
debug run: BM25_score('flew first motor driven airplan', 4)
ret value: 2.8959261945969574
$ ./run_BM25.py -s -c transport_inventions.txt BM25_score \
                "flew first motor driven airplan" 6
nlp params: ('transport_inventions.txt', './stopwords_en.txt', True)
debug run: BM25_score('flew first motor driven airplan', 6)
ret value: -0.6030241558748664
```

Task 4: In your solution, write a function called `find_top_matches` that calculates the BM25 score for a question (a string) and every document in the corpus. Both the question and the documents should have undergone pre-processing prior to the BM25 score calculation, taking into account whether stemming is enabled. The *n* top-scoring **original** documents should be returned in the form of a list of strings.

As above, pre-processed texts will be available in fields of your `NLPTasks` object.

You can verify that your function behaves correctly on the command line. For example, you should obtain the following output:

```
$ ./run_BM25.py -s -c transport_inventions.txt top_matches \
                "Who flew the first motor-driven airplane?" 3
nlp params: ('transport_inventions.txt', './stopwords_en.txt', True)
debug run: top_matches('Who flew the first motor-driven airplane?', 3)
ret value: [
  '1903: Orville Wright and Wilbur Wright flew the first motor-driven airplane.\n',
  '1967: Automatic train operation introduced on London Underground.\n',
  '2002: Segway PT self-balancing personal transport was launched
         by inventor Dean Kamen.\n']
ret count: 3
```

```
$ ./run_BM25.py -s -c transport_inventions.txt top_matches \
                "When did the world’s first underground railway open?" 3
nlp params: (’transport_inventions.txt’, ’./stopwords_en.txt’, True)
debug run: top_matches("When did the world’s first underground railway open?", 3)
ret value: [
  "1863: London’s Metropolitan Railway opened to the public
         as the world’s first underground railway.\n",
  ’1890: The City and South London Railway (C&SLR) was the first deep-level
         underground "tube" railway in the world, and the first major railway
         to use electric traction\n’,
  ’1967: Automatic train operation introduced on London Underground.\n’]
ret count: 3
```

Extension: Question Answering Integration

This part of the lab exercise will **not** be marked. However, you are strongly encouraged to also engage with this activity so that you can gain a full appreciation of how even a simple information retrieval module based on BM25 can help improve — dramatically — the answers produced by a generative large language model.

Google Colab familiarisation. Due to the fact that generative large language models are difficult to run on local machines given their required computational resources, we will make use of Google Colab which requires a Google account. It is a cloud-based platform for developing and running Python notebooks that gives you access to computational resources (such as bigger RAM and GPUs). Please explore Google Colab now if you have not done so before. Note that the model that we will use does not require you to subscribe to any of the Google Colab paid products; it will run even on a free Google Colab account.

Obtaining a Huggingface access token. Huggingface is the biggest repository of LLMs that supports the loading of models directly from code. However, this requires an access token. To obtain one, please sign up for a Huggingface account. Once you have an account, you should be able to find your access token by clicking on your profile icon, then `Settings` and finally `Access Tokens`. You will need this token as you use the Retrieval-augmented QA notebook (described below).

Retrieval-augmented QA notebook. Access our pre-prepared notebook. Create a copy of the notebook by clicking on the `File` menu and then the `Save a copy in Drive` option. Follow the cells in the notebook and observe the impact of the BM25 retrieval module on QA.

Submission

Please follow the `README.md` instructions in your `COMP24011_2023` GitLab repo. Refresh the files of your `lab4` branch and develop your solution to the lab exercise. The solution consists of a single file called `nlp_tasks.py` which must be submitted to your GitLab repo and tagged as `lab4_sol`. The `README.md` instructions that accompany the lab files include the `git` commands necessary to commit, tag, and then push **both** the commit and the tag to your `COMP24011_2023` GitLab repo. Further instructions on coursework submission using GitLab can be found in the CS Handbook, including how to change a `git` tag after pushing it.

The deadline for submission is **18:00 on Friday 8th December**. In addition, no work will be considered for assessment and/or feedback if submitted more than **2 weeks** after the deadline. (Of course, these rules will be subject to any mitigating procedures that you have in place.)

The lab exercise will be **auto-marked** offline. The automarker program will download your submission from GitLab and test it against our reference implementation. For each task the return value of your function will be checked on a random set of valid arguments. A time limit of 10 seconds will be imposed on every function call, and exceeding this time limit will count as a runtime error. If your function does not return values of the correct type, this will also count as a runtime error.

A total of 20 marks is available in this exercise, distributed as shown in the following table.

Task	Function	Marks
1	`NLPTasks.preprocess()`	5
2	`NLPTasks.calc_IDF()`	5
3	`NLPTasks.calc_BM25_score()`	5
4	`NLPTasks.find_top_matches()`	5

The marking scheme for all tasks is as follows:

- You obtain the first 0.5 marks if all tests complete without runtime errors.
- The proportion of tests with fully correct return values determines the remaining 4.5 marks.

During marking, your `NLPTasks` object will be initialised independently. This means that when functions that require text pre-processing get tested, your object will have all its fields initialised with correct values independent of your implementation of Task 1.

☞ In addition to the two corpora provided in your repo, your solution will be tested with a hidden corpus for marking. This will only be released together with the results and feedback for the lab.

Important Clarifications

- It will be very difficult for you to circumvent time limits during testing. If you try to do this, the most likely outcome is that the automarker will fail to receive return values from your implementation, which will have the same effect as not completing the call. In any case, an additional time limit of 300 seconds for all tests of each task will be enforced.
- This lab exercise is fully auto-marked. If you submit code which the Python interpreter does not accept, you will score 0 marks. The Python setup of the automarker is the same as the one on the department's Ubuntu image, but only a **minimal** set of Python modules are available. If you choose to add `import` statements to the sample code, it is **your responsibility** to ensure these are part of the default Python package available on the lab machines.
- It doesn't matter how you organise your `lab4` branch, but you should avoid having multiple files with the same name. The automarker will sort your directories alphabetically (more specifically, in ASCII ascending order) and find submission files using breadth-first search. It will mark the first `nlp_tasks.py` file it finds and ignore all others.
- Every file in your submission should only contain printable ASCII characters. If you include other Unicode characters, for example by copying and then pasting code from the PDF of the lab manuals, then the automarker is likely to reject your files.