COL 764 Assignment 1 - Exploring the Impact of Document Preprocessing and Tokenization on Retrieval Performance

2024 - Version 1.0
(dated 19th August 2024)

Deadline

Submission of the complete implementation, and the report on the algorithms is due on **September 3, 2023, 11:59 PM**. All submissions are to be made on Moodle.

Instructions

Follow all instructions. Submissions not following these instructions will not be evaluated.

1. This programming assignment is to be done by each student individually. Do not collaborate by sharing code, algorithm, and any other pertinent details with each other. You are free to discuss and post questions to seek clarifications. Please post all discussions related to this assignment under **"Assignment 1"** channel on Teams.
2. All programs have to be written either using **Python/Java/C/C++** programming languages only. Anything else requires explicit prior permission from the instructor. The machine we use to evaluate your submissions has the following versions:
   - **C**, **C++**: gcc12.3;
   - **Java**: Java (SE) 22 (we will not use OpenJDK);
   - **Python:** version 3.12.

   We recommend you to use same versions to avoid the use of deprecated/unsupported functions or take care to ensure required compatibility.
3. A single zip of the source code has to be submitted. The zip file should be structured such that
   - upon deflating all submission files should be under a directory with the student's registration number. E.g., if a student's registration number is `20XXCSXX999` then the zip submission should be named 20XXCSXX999.zip and upon deflating **all contained files** should be under a directory named ./20XXCSXX999 only (names should be in uppercase). Your submission might be rejected and not be evaluated if you do not adhere to these specifications.
   - apart from source files, the submission zip file should contain a build mechanism *if needed* (allowed build systems are Maven, Ant and Gradle for Java, Makefile for C/C++). It is the responsibility of each student to ensure that it compiles and generates the necessary executable as specified. **Please include an empty file in case building is not required** *(e.g. if you are using Python)*.
   - Do not submit notebooks – all programs will be run from the commandline only.
   - the list of files to be submitted, along with their naming conventions and call signatures as given in the instructions below.
4. **Note that we will use only Ubuntu Linux machines to build and run your assignments.** So take care that your file names, paths, argument handling etc. are compatible.

5. You *should not* submit data files. If you are planning to use any other "special" library, please talk to the instructor first (or post on Teams).
6. Note that your submission will be evaluated using larger collection. Only assumption you are allowed to make is that all documents will be in English, have sentences terminated by a period, and all documents are in one single directory. The overall format of documents will be same as given in your training. The collection directory will be given as an input (see below).
7. **Note that there will be no deadline extensions. Apart from the usual "please start early" advise, I must warn you that this assignment requires significant amount of implementation effort, as well as some 'manual' tuning of parameters to get good speed up and performance. Do not wait till the end.**

1 Assignment Description

The goal of this assignment is to explore the impact of document preprocessing and tokenization choices on the overall retrieval performance for an English corpora. As a side-effect you are also required to implement an efficient-to-query inverted index structure (from scratch).

1.1 Input

You are given a document collection – extracted from a benchmark collection in TREC, consisting of English documents. Each document has a unique document id, and the overall document is represented as a JSON fragment (see the colored box below). All content *except the doc_id* can be indexed, and this **indexable portion** of the document will be referred to as 'document content' in the rest of this document. The indexing is *full text* – i.e., you should not be doing any stop-word elimination or stemming on the document collection.

The document collection itself is specified as a single JSON file, containing one or more documents (each is a JSON fragment).

An Example of JSON Fragment Representing a Document.

Each document is enclosed in curly braces, contains a "doc_id" field, and remaining contents under different (collection specific) set of JSON fields.

```
{"doc_id": "2b73a28n",
 "title": "Role of endothelin-1 in lung disease",
 "doi": "10.1186/rr44",
 "date": "2001-02-22",
 "abstract":
   "Endothelin-1 (ET-1) is a 21 amino acid peptide with diverse biological activity
    that has been implicated in numerous diseases. ET-1 is a potent mitogen regulator
    of smooth muscle tone, and inflammatory mediator that may play a key role in
    diseases of the airways, pulmonary circulation, and inflammatory lung diseases,
    both acute and chronic. This review will focus on the biology of ET-1 and its
    role in lung disease."
}
```

Except the doc_id field, you are free to use the contents of all other fields for indexing. You may safely assume that all documents are well-formed (i.e., no broken JSON fields/formats), and contain no arbitrary non-ASCII characters.

1.2 Corpus Details

The training corpus is available in HPC for download at the following path:
`/scratch/cse/faculty/srikanta/COL764-A1-2024/train_data`

qrels_test.json format

```
{"query_id":  "1", "doc_id":  "021q9884", "relevance":  1, "iteration":  "4"}
{"query_id":  "1", "doc_id":  "02f0opkr", "relevance":  1, "iteration":  "1"}
{"query_id":  "1", "doc_id":  "047xpt2c", "relevance":  0, "iteration":  "3.5"}
```

where

1. **query_id** is the Query ID,
2. **doc_id** is the official document id that corresponds to the ”doc_id” field in the documents,
3. **relevance** is a binary code of 0 for not relevant and 1 for relevant, and
4. **iteration** is the feedback iteration (not used).

The order of document ids for each query in the qrels_test file is not indicative of relevance or degree of relevance. Only a binary indication of relevant (1) or non-relevant (0) is given. Documents not occurring in the qrels file were not judged by the human assessor and are assumed to be non-relevant in the evaluations.

test_query.txt format

```
{"query_id": "1", "title": "coronavirus origin",
 "description": "what is the origin of COVID-19",
 "narrative": "seeking range of information about the SARS-CoV-2 virus's origin,
 including its evolution, animal source, and first transmission into humans"
}
{"query_id": "2", "title": "coronavirus response to weather changes",
 "description": "how does the coronavirus respond to changes in the weather",
 "narrative": "seeking range of information about the SARS-CoV-2 virus viability
 in different weather/climate conditions as well as information related to transmission
 of the virus in different climate conditions"
}
```

With in each query, the following JSON fields are present:

1. `query_id:` Encloses the query `( also known as topic )` identifier. Note that the query identifier here is not necessarily an integer.
2. `title:` (optional) query title (a short title of the query)
3. `description:` the description of the query.

You are expected to make use of content under `description` and `title` fields for formulating your queries. Please note that you need not make use of both the fields.

1.3 Task

This assignment consists of the following three subtasks:

1. Implement 3 different tokenizers which will be used for inverted index construction:
   (a) **Simple tokenizer**: Use `<white-space>` and `,.:;"’` as the set of punctuation, as delimiter, for tokenization.
   (b) **BPE tokenizer**: While there are many implementations and pre-trained BPE tokenizers, you are expected to implement BPE tokenizer yourself. You may assume that tokens don’t span word boundaries detected in the simple tokenizer.

(c) **WordPiece tokenizer:** We did not discuss WordPiece tokenizer explicitly in the class, but you can learn about it from HuggingFace tutorial page. Implement WordPiece tokenizer from scratch (i.e., it will compute the vocabulary to index on). Similar to BPE, you may assume that tokens don't span word boundaries detected in the simple tokenizer.

For both BPE and WordPiece you should use a 50% sample of the given corpus as the training corpus.

Choosing Tokenizer Training Corpus: If your entry number ends in an ODD digit then you select the **first 50%** documents in the corpus, otherwise (i.e., if your entry number ends in an EVEN digit) then select the **last 50%** documents in the corpus. Do not change the sequence of documents in the docs file.

A Word of Caution

Note that you will be tempted to simply use the BPE and WordPiece tokenizers available on the web (e.g., from HuggingFace itself). However, that will not earn any points to you in this assignment – you are expected to implement them by yourself. You can use the HuggingFace tokenizers for debugging your implementation of BPE and WordPiece tokenizers.

2. Invert the document collection and build an on-disk inverted index, consisting of (at least) (i) a dictionary/vocabulary file and (ii) a single file containing postings lists of all the index-terms. The document inversion should be capable of using any one of the three tokenizers you have implemented above. You should not perform any stop-word elimination and stemming.

3. You must provide a ranked retrieval system using TF-IDF scoring using the index you have constructed above.

   (a) **Term Frequency (TF)**: We will use
   $$tf_{ij} = 1 + \log_2(f_{ij})$$
   as the term-frequency component, where f_{ij} is the number of times the index-term i appears in document d_j.

   (b) **Inverse Document Frequency (IDF)**: We will use
   $$idf_i = \log_2\left(1 + \frac{N}{df_i}\right)$$
   where df_i is the number of documents that the index-term i appears in, and N is the total number of documents in the collection.

   Your program will have to take as input a large list of keyword queries, and output the results to a separate file. The specific format of the output file is described later on in this document. Note that the queries are to be evaluated in exactly the same order as given.

1.4 Metrics of Interest

Apart from functional implementation of the overall task, you should report the following metrics on the given document collection:

Performance of Ranked Retrieval: You will required to generate up to 100 best results for each query. We will use $F_1@\{100, 50, 20, 10\}$ as the metric to evaluate how well your retrieval performance is. We expect the performance of all submissions for the case of normal tokenizer to similar.

Retrieval Efficiency: Average time taken per query (including loading of list(s), dictionary, tokenization of queries as required) from the time the query is passed to the system to the time all (top-100) results are retrieved. It is computed as follows:
$$Efficiency = \frac{|T_Q|}{Q}$$

where T_Q is the time taken for answering (and writing the results to the file) for ***all*** given queries, and Q is the total number of queries.

1.5 Program Structure

In order to achieve these, you are required to write the following programs:

1. **Tokenizer and Dictionary Construction:** This program should be named as `dict_cons.{py|c|cpp|C|java}` and should generate a single executable. It is called as via a shell script (that you should submit) as follows:
   `dictcons.sh [coll-path] {0|1|2}`
   where,

   `coll-path` specifies the absolute path of the directory containing the training documents (in the same JSON format as the corpus) for the tokenizer (you must use the entire set of documents here),
   `{0|1|2}` specifies the name of the tokenizer used by the program – **`0`** specifies the simple tokenizer, **`1`** specifies the BPE tokenizer, and **`2`** specifies the WordPiece tokenizer.

   The program should generate **one** file:
   `output.dict` containing the tokens learnt by the tokenizer (i.e., list of white-space terminated terms for simple tokenizer, BPE vocabulary by the BPE tokenizer, and Wordpiece vocabulary by the Wordpiece tokenizer).

2. **Inverted indexing of the collection:** Program should be named as `invidx_cons.{py|c|cpp|C|java}`.
   It will be called via a shell script as follows:
   `invidx.sh [coll-path] [indexfile] {0|1|2}`
   where,

   `coll-path` specifies the absolute path of the directory containing the files containing documents of the collection, and
   `indexfile` is the name of the index files that will be generated by the program
   `{0|1|2}` specify the name of the tokenizer used by the program – **`0`** specifies the simple tokenizer, **`1`** specifies the BPE tokenizer, and **`2`** specifies the WordPiece tokenizer.

   `invidx.sh` is a BASH shell-script that will be a wrapper for your program. Do not use this script for building/compiling your assignment (which should be done using `build.sh`) .

   The program should generate **two** files:

   (a) **`indexfile.dict`** contains the dictionary and
   (b) **`indexfile.idx`** contains the inverted index postings. Note that it is expected to be a *binary file* (not printable text) containing the sequence of document identifiers for each postings list. Further, it can also contain –if needed– a mapping between document identifier given in the collection, and the document number used in the index. There is no restriction on how these (and any other) info will be stored in the postings list file.

   Construct an appropriate inverted index consisting of postings-list and dictionary structure which are stored on disk in the specified filenames.

3. **Search and rank:** Your submission should also consist of another program called `tf_idf_search.sh` for performing TF-IDF retrieval using the index you have just built above. It will be called via a shell script as follows:

```
tf_idf_search.sh [queryfile] [resultfile] [indexfile] [dictfile]
```

`queryfile` a full path of the file containing queries in the same format as given in the test_query.txt.

`resultfile` the full path and name of the output file `resultfile` which is generated by your program, following format (see below)

`indexfile` the full path to the index file generated by `invidx_cons` program above which should be used for evaluating the queries

`dictfile` the full path to the dictionary file generated by the `invidx_cons` program above which should be used for evaluating the queries

Any other additional information (e.g., compression technique used if any) must be stored as part of index file.

Result Output Format: We will adopt the format used by the **trec_eval** (`https://github.com/usnistgov/trec_eval`) tool for verifying the correctness of the output using a program. Therefore, it is **very** important that you follow the format exactly as specified below. The result file format instructions are given below:

Output Format

Lines of results_file are of the form

```
qid    iteration       docid          relevancy
Q          0        SZF08-175-870          1.0
```

giving document id (a string, as given in the collection) retrieved by query qid ('Q') with relevancy (i.e., the TF-IDF score as a floating point number). It is expected that for each qid, the docno's are sorted in decreasing order of their relevancy values. The result file may not contain NULL characters.

1.6 Submission Plan

All your submissions should strictly adhere to the formatting requirements given above. You might choose to split your code by creating additional files/directories but it is your responsibility to integrate them and make them work correctly. You can also generate temporary files at runtime, if required, **only within your directory**.

- You should also submit a README for running your code, and a PDF document containing the implementation details as well as any tuning you may have done. **Name them** README.{txt|md} **and** 20XXC-SXX999.pdf **respectively**. The PDF report should also contain details of how the experiments were conducted, what the results are –speed of construction, performance on the released queries, query execution time etc.

- Only `BeautifulSoup`, `lxml`, `nltk`, and libraries distributed with python (e.g. `re`) will be available in the Python evaluation environment. **You should not include the source code of these libraries directly in your submission. All allowed dependencies of libraries that you require should be handled in your build script. You can assume that there is internet connectivity during the build.**

- You are also required to share the details of directories on HPC, that contain Index and Dictionaries that you built.

1.7 Evaluation Plan

- The set of commands for evaluation of your submission roughly follows -

Tentative Evaluation Steps (which will be used by us)

```
$ unzip 20XXCSXX999.zip
$ cd 20XXCSXX999
$ bash build.sh
$ bash dictcons.sh <arguments>
$ bash invidx.sh <arguments>
$ bash tf_idf_search.sh <arguments>
```

- Note that all evaluations will be done on not only the queries that are released, but on a set of held-out queries that are not part of the training set. We will use a different query file than test_query.txt given in the shared folder, although the format of the query file will remain the same. Thus, your implementation should be able to handle new terms in the query.

It is important to note that instructor / TA will not make any changes to your code – including changing hardcoded filenames, strings, etc. If your code does not compile and/or execute as expected, you will be called for a meeting with the TA where you can make minor changes to the code. **There will be a penalty of 10% of the overall marks for such a case – however minor the change is going to be.** It is your responsibility to guard against this.

There are timeouts at each stage exceeding which the program will be terminated –irrespective of whether it has completed or not. You may assume the following timeouts:

Stage	**timeout**
tokenization and dictionary construction (`dictcons.sh`)	5 minutes
inverted index construction (`invidx.sh`)	30 minutes
searching (`tf_idf_search.sh`)	15 minutes (for 25 queries)

These timeouts have been estimated as per the platform we use for all evaluations and a model implementation.

1.8 Tentative breakup of marks assignment

In general, a submission qualifies for evaluation if and only if it adheres to the specifications given above (arguments, structure, use of external libraries, correct output format, input format adherence, etc.). Given this requirement, the marks assignment for correct implementation of:

basic inverted index (postings list + dictionary)	20
BPE tokenizer	15
Simple tokenizer	10
WordPiece Tokenizer	15
Ranked retrieval performance	15
Retrieval efficiency	15
Report	10
Total	100

Note that in ranked retrieval performance, and retrieval efficiency, the evaluation is relative. The top-3 will get full marks, next 5 will get 90% of marks, and so on.