

Programming Assignment: Programming Assignment 1

✓ Passed · 105/125 points

Deadline The assignment was due on March 24, 11:59 PM PDT
You can still pass this assignment before the course ends.

Instructions

My submission

Discussions

MeTA Overview

Before starting the assignment, you are highly encouraged to read [MeTA's system overview](#) to gain a high level understanding of how MeTA operates. Throughout the assignments, if you want to know more about a certain class or function, you can use the search toolbar in [MeTA's documentation](#), which provides a brief explanation of the different modules.

If you have questions about the programming assignment, use the [Programming Assignments Forum](#). This is a great place to ask questions and also help your fellow classmates.

Downloading the Assignment

In what follows, we assume that you have already installed MeTA and the directory **meta** is located in **~/Desktop/**.

1. Download Assignment_1.tar.gz (below) and extract it into the parent directory of **meta**.
For example, if **meta** is in **~/Desktop/**, then you should extract the **assignment** in **~/Desktop/**
2. In the terminal, change the directory to **Assignment_1** and run the bash script **Setup.sh**.
This can be done using the following two commands:

```
1 cd Assignment_1/  
2 ./setup.sh
```

3. Now using MeTA as a library, to compile:

```
1 cd Assignment_1/build
2 cmake .. -DCMAKE_BUILD_TYPE=Release; make -j8
```

Download Assignment_1.tar.gz:

Assignment_1.tar.gz

Warm Up!

In this section you will start by exploring some of the basic text processing techniques such as tokenization, stemming, and stopword removal. These techniques are usually applied to the corpus prior to the indexing stage. You will also perform part-of-speech tagging on a text document.

Tokenization is the process of segmenting a stream of words (such as a document) into a sequence of units called tokens. Loosely speaking, when tokenization is performed on the word level, the tokens will be the words in the document. To perform tokenization, MeTA relies on the default segmentation boundaries between words and sentences defined by the Unicode Standard (see [Unicode Standard Annex #29](#) for more information).

Typically, after tokenization is performed, a number of text filtering techniques are applied on the tokens in order to perform more effective and efficient retrieval. These techniques include:

- **Stopword removal:** Stopwords are frequent words that are not informative for the task at hand. For most retrieval applications, words such as "in," "or," "have," and "the" are not useful for identifying the relevant documents, and thus, they are discarded before the indexing stage. This considerably helps in reducing the size of the inverted index since stopwords occur very frequently and tend to have large postings lists. MeTA uses a list of stopwords saved in **/meta/data/lemur-stopwords.txt**. Feel free to open the file and have a look at some of the popular stopwords.
- **Conversion to lower case:** For most applications, converting all letters to lower case can help in boosting the retrieval performance. The intuition here is that uppercase and lowercase forms of words usually refer to the same concept and should not be treated as orthogonal dimensions. However, this conversion can lead to inaccuracies in certain situations. For example, a proper noun like "CAT" (a construction company) will have the same representation as the common noun "cat."

- **Stemming:** It is the process of converting words back to their original stems or roots. For example, the words "retrieve," "retrieval," and "retrieving" will all be mapped to the same root "retrieve." This can prevent different forms of the same word from being treated as orthogonal concepts. MeTA uses a very well known stemmer called the Porter2 English Stemmer; see [English \(Porter 2\) stemming algorithm](#) for more information on how this stemmer operates.

Now you are ready to apply these concepts using MeTA!

Task 1: Stopword Removal (5 pts)

In this task you will run tokenization and stopwords removal on a document. Go to **Assignment_1/build/Assignment1/**. You should find a text document called **doc.txt**. Open **doc.txt** and have a look at its content. The document contains the description found on [the main page of this course](#).

In the terminal, perform the following:

```
1 cd Assignment_1/build
2 cmake .. -DCMAKE_BUILD_TYPE=Release; make -j8
3 ./analyze ../config.toml Assignment1/doc.txt --stop
```

After running the above, if everything is setup correctly, then you should see:

```
1 $ ./analyze ../config.toml Assignment1/doc.txt --stop
2 Running stopwords removal ->
3 file saved as Assignment1/doc.stops.txt
```

Open the output file **Assignment_1/build/Assignment1/doc.stops.txt** and you should see how tokenization and stopwords removal have been applied.

In the 2nd command above, you have executed a compiled program called **analyze** and passed to it a configuration file called **config.toml**, the document's path, and the parameter "stop," which tells the program to remove stopwords. The **analyze** is a demo program that supports multiple text analysis functions. Passing **config.toml** to programs in MeTA is a default practice as it gives the program all the required parameters to run.

Open **Assignment_1/config.toml** and examine it. You should see a list of configuration parameters, including the directory of the list of stopwords (which should appear in the first line). If you have a new list of stopwords, you can simply point MeTA to it (but do not change the list for this assignment).

After finishing Task 1, you should submit the output file using the submission script. In the shell, execute:

```
1 cd Assignment_1/build/Assignment1
2 wc < doc.stops.txt | sed 's/^\s*//g' > doc.stops.txt.wc
```

Submit the file **docs.stops.txt.wc** to Task 1. In the *My submission* tab, click + *Create submission*. Then click on **Task 1: Stopword Removal** and upload your file.

Task 2: Stemming (10 pts)

Now you will perform tokenization and stemming on the same document **doc.txt**. The **analyze** program you ran in Task 1 provides several other text analysis functionalities, including stemming. For the different functions implemented in **analyze**, open **Assignment_1/analyze.cpp** and have a look. In addition, simply run **./analyze** and you should see:

```
1 $ ./analyze
2 Usage: ./analyze config.toml file.txt [OPTION]
3 where [OPTION] is one or more of:
4 --stem perform stemming on each word
5 --stop remove stopwords
6 --stopstem remove stopwords and perform stemming
7 --pos annotate words with POS tags
8 --pos-replace replace words with their POS tags
9 --parse create grammatical parse trees from file content
10 --freq-unigram sort and count unigram words
11 --freq-bigram sort and count bigram words
12 --freq-trigram sort and count trigram words
13 --all run all options
```

Your task is to perform stemming on **doc.txt** using the **analyze** program.

After running stemming, you should see a new file named **doc.stems.txt** in output directory **Assignment_1/build/Assignment1/**. Examine the file and see how words are mapped back to their stems.

Now do the word count again and

```
1 wc < doc.stems.txt | sed 's/^\s*//g' > doc.stems.txt.wc
```

Submit your output file **doc.stems.txt.wc**. In the *My submission* tab, click + *Create submission*. Upload your file to **Task 2: Stemming**.

Task 3: Part-of-Speech Tagging (10 pts)

As you have learned in the lecture on natural language processing, part-of-speech tagging is the process of assigning parts of speech (such as verb, noun, adjective, adverb, etc.) to the words in a given text. Your task is to do POS tagging (without replacement) on **doc.txt** using

the **analyze** program. After executing the program, examine the output in **doc.pos-tagged.txt**. You should see how a POS tag is assigned to each word after the underscore. Note that the tags are abbreviated; for example, the tag "NN" stands for a noun. For a list of commonly used tags and their meanings see the UPenn Treebank list. For a more comprehensive covering of POS tagging, you can check MeTA's POS Tagging Tutorial. Don't forget to submit your output file after running the following!

```
1 wc < doc.pos-tagged.txt | sed 's/^\s*//g' > doc.pos-tagged.txt.wc
```

Task 3.5: Writing a New Function (15 pts)

Now that you have explored some of the basic text analysis techniques in **analyze**, you will add a new function to **analyze** that does stopwords removal followed by stemming on the same text document.

MeTA implements the different text analysis techniques such as stemming and stopwords removal as filters that can be chained together. See [MeTA's filters Reference](#) for a list of the different supported filters.

To implement your new function, edit **Assignment_1/analyze.cpp**. Examine and try to understand how the functions **stem** and **stop** are performing tokenization and filtering.

Find the function:

```
1 void stopstem(const std::string& file,  
2               const cptoml::table& config)
```

Your task is to complete the function in order to perform the required task. In the place where we wrote:

```
1 // Insert the line required to do stopwords removal here  
2 // Insert the line required to do stemming here (using Porter2 Stemmer)  
3
```

You should only add the stopwords removal filter and the Porter2 stemming filter. (2 Lines in total) Do not add any other filters.

Since source code is changed, you should recompile after writing your code:

```
1 cd Assignment_1/build  
2 cmake .. -DCMAKE_BUILD_TYPE=Release; make -j8  
3 ./analyze ../config.toml Assignment1/doc.txt --stopstem  
4
```

and we still ask you to run

```
1 wc < doc.stopstem.txt | sed 's/^\s*//g' > doc.stopstem.txt.wc
```

and submit your result.

Build the Search Engine

In this section, you will perform and experiment with indexing and retrieval on a special dataset called the MOOCs dataset.

Exploring the Dataset

The MOOCs dataset contains the descriptions found on the webpages of around 23,000 MOOCs (Massive Open Online Courses). You will start by exploring the dataset. Navigate to **Assignment_1/moocs**. This directory contains the files that describe the dataset.

- **moocs.dat** contains the content of the webpages of all the MOOCs; each MOOC's main page occupies exactly one line in the file. Feel free to open the file to get an idea about the contents, but be wary that it is a large file and might take some time to load.
- **moocs.dat.names** contains the names and the URLs of the MOOCs. The entry on line x in **moocs.dat.names** corresponds to the MOOC on line x in **moocs.dat**.
- **moocs-queries.txt** contains a set of queries that you will use to evaluate the effectiveness of your search engine.
- **moocs-qrels.txt** contains the relevance judgments corresponding to the queries in **moocs-queries.txt**. Each line in **moocs-qrels.txt** has the following format: (querynum documentID 1). This means that the document represented by documentID is a relevant document for the query whose number is querynum. The relevance judgments in **moocs-qrels.txt** were created by human assessors who ran the queries and chose the relevant documents. Later on in the assignment, you are going to use these judgments to quantify the performance of your search engine.

Indexing

Now that you have an idea about tokenization, text filtering, and the dataset to be used, you will proceed to index the MOOCs dataset. In this process, an inverted index will be created. As you have seen in the lectures, the inverted index is a data structure that supports efficient retrieval of documents by allowing the lookup of all the documents that contain a specific term.

Before you proceed to index the dataset, we encourage you to open **config.toml** and examine the settings that we have already configured. For instance, the snippet shown below tells the indexer where to find the MOOCs dataset, specifies that it is a line corpus (i.e., each document is on one line), and defines the name of the inverted index to be created. The forward index will not be used in this assignment.

```
1 query-judgements = "../../meta/data/moocs/moocs-qrels.txt"
2 querypath = "../../meta/data/moocs/"
3 corpus = "line.toml"
4 dataset = "moocs"
5 forward-index = "moocs-fwd"
6 inverted-index = "moocs-inv"
```

You should also have a look at another important snippet:

```
1 [[analyzers]]
2 method = "ngram-word"
3 ngram = 1
4 filter = "default-unigram-chain"
```

The settings under the analyzers tag control how tokenization and filtering are performed, prior to creating the inverted index. The tokenizer will segment each document into unigrams. MeTA uses its default filtering chain, which performs a couple of predefined filters including lower case conversion, length filtering (which discards tokens whose length is not within a certain range), and stemming. To read more about modifying META's default tokenization and filtering behavior see [MeTA's analyzers and filters](#) page.

To index the dataset, in the **Assignment_1/build** directory run:

```
1 ../../meta/build/index ../../config.toml
```

(index program is in the **meta/build** directory under root). You should see something like below:

```
1 $ ./meta/build/index ../../config.toml
2 1463154868: [info] Loading index from disk: moocs-inv (root/meta/src
   /index/inverted_index.cpp:178)
3 Number of documents: 23566
4 Avg Doc Length: 383.014
5 Unique Terms: 193614
6 Index generation took: 0.007 seconds
```

This will start by performing tokenization and applying the text filters defined in **config.toml**; it then creates the inverted index and places it in **meta/build/moocs-inv**. When the program finishes execution, you should get a summary of the indexed corpus as above

Searching

After creating the inverted index, you can efficiently search the MOOCs dataset. The ranking function to be used in retrieving documents is defined in **config.toml**. Open **config.toml** and look for the **ranker** tag. Under this tag, you will see that the default ranker is "bm25" along with values assigned to its three parameters. To test your search engine, you can use the **interactive-search** program provided with MeTA:

```
1 cd Assignment_1/build
2 ../../meta/build/interactive-search ../config.toml
```

Enter any query you want, and the top results should show up instantaneously. Feel free to experiment with different queries to explore the contents of the dataset.

When you finish, enter a blank query to exit.

In what follows you will evaluate the performance of the search engine using different ranking functions and parameters. The main evaluation measure to be used is MAP (mean average precision), which is the arithmetic mean of the average precision of all the queries being used for evaluation. We have provided you with a program called **ranking-experiment** which evaluates the MAP of any retrieval function you specify in **config.toml**.

Task 4: BM25 (5 pts)

Evaluate the performance of bm25 with default parameters (i.e., do not change **config.toml**) by running:

```
1 cd Assignment_1/build
2 ./ranking-experiment ../config.toml task4
3
```

You should see a list of the top results corresponding to the different test queries, the precision at 10 for each query, and the final MAP value.

At the same time, we create a output file for you to submit:

/Assignment_1/build/Assignment1/task4.txt

Submit **task4.txt**.

Task 5: BM25 with tuned parameters (10 pts)

Change the parameter **b** of bm25 in **config.toml** to 0.75


```
1  ranker]
2  method = "bm25"
3  k1 = 1.2
4  b = 0.75
5  k3 = 500
```

and run again:

```
1  cd Assignment_1/build
2  ./ranking-experiment ../config.toml task5
3
```

and submit your **/Assignment_1/build/Assignment1/task5.txt**

Task 6: PL2 (30 pts)

MeTA has several built-in rankers other than bm25; see MeTA's Index Reference page and the directory meta/src/index/ranker for the list of built-in rankers. In this task you will implement a well known retrieval function called PL2, which is not available in MeTA. PL2 is a member of the Divergence from Randomness framework, which is based on the idea that the more different the within-document term frequency is from the term frequency generated by a random process, the higher the information content of the term is. PL2's scoring function is given by:

$$\text{Score}(Q, D) = \sum_{t \in Q \cap D} \frac{1}{tf_n + 1} \left(tf_n \log_2 \frac{tf_n}{\lambda} + \left(\lambda + \frac{1}{12tf_n} - tf_n \right) \log_2 e + \frac{1}{2} \log_2(2\pi tf_n) \right)$$

where

$$tf_n = tf \log_2 \left(1 + c \frac{Avg_l}{D_l} \right)$$

and tf is the term frequency in the document, Avg_l is the average document length in the whole corpus, D_l is the document length, and λ and c are positive tunable parameters.

Every ranking function in MeTA should have its own class that subclasses the base class **ranker**. Go to [MeTA's ranker Class Reference](#) and have a quick look at the member functions of **ranker**. One important function in **ranker** you should be aware of is:

```
1 float score_one(const index::score_data&) override;
```

which calculates the ranking score of one matched term. That is, if the scoring function is written as

$$\text{Score}(Q, D) = \sum_{t \in Q \cap D} w(t, Q, D)$$

where t is a matched term and $w(t, Q, D)$ is the score of that term, then **score_one** = $w(t, Q, D)$. After each call to **score_one**, the result will be added to the document's accumulator, which is responsible for returning the final document score.

The struct of type **score_data** passed as a parameter to **score_one** contains important statistics about the matched term (such as the number of occurrences in the document and the document length) and other global statistics like the average document length in the corpus. For more information on the members of this struct, check [MeTA's score_data Reference](#).

Now that you have a basic understanding of the base class **ranker** and the function **score_one**, you should be able to implement the PL2 ranking function. Go to **meta/src/index/tools/** and open **ranking-experiment.cpp** in a text editor or IDE. We have already provided a code that creates a new ranker class called "pl2_ranker." As you can see, "pl2_ranker" is a derived class from the base class **ranker**. Examine the members of "pl2_ranker" and the comments next to them.

Your task is to complete the implementation of the function:

```
1 float score_one(const index::score_data&)
```

in order to return the score defined by PL2. Make sure to follow the instructions provided as comments in the body of the function.

In addition, in config.toml, change the ranker to "pl2" with appropriate parameters. In below, we use "#" to comment out lines that we don't need.

```
1 [ranker]
2 #method = "bm25"
3 #k1 = 1.2
4 #b = 0.75
5 #k3 = 500
6 method = "pl2"
7 c = 7
8 lambda = 0.1
```

Now, you need to recompile the code again:

```
1 cd Assignment_1/build
2 cmake .. -DCMAKE_BUILD_TYPE=Release; make -j8
3 ./ranking-experiment ../config.toml task6
```

And again, submit your results **/Assignment_1/build/Assignment1/task6.txt**

Task 7: Tuning PL2 (20 pts)

Note: If you are a Mac user, you will need to use the GNU utils (e.g., gnu sed) instead of BSD to get the results.

In this task you will tune the PL2 ranking function in order to get better values of c and λ . Open **ranking-experiment.cpp**, and look for the function "pl2_tune." This function is expected to sweep over pairs of the parameters c and λ and select the pair that gives the highest MAP. However, the function has a few incomplete lines:

```
1 maxmap = 0; // Change 0 to the correct value
2 cmax = 0; // Change 0 to the correct value
3 lambdamax = 0; // Change 0 to the correct value
```

Read the comments beside the function's code and try to understand how it works. After that, change the values in the incomplete lines.

You should now recompile MeTA and run **ranking-experiment**:

```
1 cd Assignment_1/build
2 cmake .. -DCMAKE_BUILD_TYPE=Release; make -j8
3 ./ranking-experiment ../config.toml task7
```

ranking-experiment will then return the maximum MAP along with the optimal parameters λ and c that achieve this maximum. You should see a noticeable improvement in the MAP over what you got in Task 6.

And again, submit your results **/Assignment_1/build/Assignment1/task7.txt**

After you finish Task 7, open **config.toml** and revert to bm25 as the default ranker. You can do this by removing the hash symbols preceding bm25 and its parameters and adding hash symbols to pl2 and its parameters.

Task 8: Relevance Judgments (20 pts)

The next programming assignment will be a competition where your search engine will be evaluated based on relevance judgments made by all the students in this course. Relevance judgments are query-document pairs marked as relevant or irrelevant by users. We expect to collect a large and diverse pool of judgments, which should allow a fair evaluation of your search engines in the second assignment.

For this purpose, you are required to come up with a query about a topic you would like to study in an online course. The query can be something like "information retrieval." You should also write a brief description of what you expect to learn from the course you are searching for. For example, one description is "To learn how to efficiently do retrieval from large datasets." In the terminal, execute:

```
1 cd Assignment_1/build
2 ./relevance-judgements ../config.toml
```

You should perform the following steps after running **relevance-judgements**:

1. Enter the query of your choice.
2. Enter a description of what you are expecting to learn. You will be shown a list of the top 20 MOOCs corresponding to your query.
3. Enter the numbers of the relevant MOOCs separated by spaces.

In case you could not find any relevant documents for your query, try another one. Also, feel free to submit more than one query as this should help more in the competition.

The relevance judgments you came up with are saved in a file called **task8.txt**. You can take a look at its content, but do not modify it.

Currently the grade cannot grade anything that has multiple lines and please remove any "return line" ('\n') from task8.txt for submission purpose. You do not add any new whitespaces.

Submit your **/Assignment_1/build/Assignment1/task8.txt** Submit the file docs.stops.txt.wc to partTask 1. In the My submission tab, click Createsubmission. Then click on Task 1: Stopword Removal and upload your file.

How to submit

When you're ready to submit, you can upload files for each part of the assignment on the "My submission" tab.

