CS246 Project 1

Overview

In this homework, we will use Elasticsearch, a popular open-source text search engine, to index a Wikipedia dataset to enable keyword search on the dataset. We will also apply various text transformations to the dataset during indexing and observe the effect of the transformations in search results.

System Setup

Follow our VirtualBox Setup Instruction to set up the virtual machine that we will use for this project. Please make sure that

1. the shared folder is working correctly and
2. you can ssh into the virtual machine through port 2462

as described in the instruction page.

All software tools needed for this project have already been installed in the virtual machine, including Elasticsearch. To start Elasticsearch in the virtual machine, enter the following command:

```
sudo service elasticsearch start
```

(You will need to start Elasticsearch using the above command, whenever you power up your virtual machine in the future.)

Wait a few seconds while Elasticsearch is stating. Once Elasticsearch has started, you can test that your Elasticsearch node is running by sending an HTTP request to port 9200 on localhost:

```
curl -XGET 'localhost:9200/?pretty'
```

which should give you a response like this:

```
{
  "name" : "Cp8oag6",
  "cluster_name" : "Elasticsearch",
  "cluster_uuid" : "AT69_T_DTp-1qgIJlatQqA",
  "version" : {
    "number" : "5.6.2",
    "build_hash" : "f27399d",
    "build_date" : "2016-03-30T09:51:41.449Z",
    "build_snapshot" : false,
    "lucene_version" : "6.6.1"
  },
  "tagline" : "You Know, for Search"
}
```

Task 1: Brush Up Your Java Skill by Implementing a "Crawler"

The primary goal of our first task it to brush up your Java programming knowledge. In this task, you will have to build a simple "Web crawler" in Java. When given a particular url, your crawler will retrieve the webpage and print the html text of the webpage to the console. If you are new to Java or if it has been a while since your last Java programming, first read A Crash Course from C++ to Java. This excellent tutorial explains the basics of Java, including how you can name, compile, and run your Java program. (It is okay to skip the parts on BlueJ in the tutorial, since we will not be using it.) If you are quite familiar with Java, but you just want to brush up on minor details quickly, you may want to read slides on Java instead. All basic tools needed for Java programming (e.g., `javac` and `java`) are available on our VM.

Now download the project1.zip file, which includes our skeleton Java code, `GetWebpage.java`. The provided code takes a URL as the command-line parameter and simply prints the URL to console. Your task is to insert your own code to the file to implement a simple Web crawler.

Your Java program should satisfy the following requirements:

1. Your program should be implemented as a single Java class GetWebpage. Your program should take the url as the first command line parameter. For example, a user should be able to execute your program like

   ```
   java GetWebpage http://stackoverflow.com
   ```

2. Given a URL, your program must display the entire html webpage on the console. For example, running the program with the url â€œhttp://stackoverflow.comâ€ should display the following:

   ```
   <html><head><title>Object moved</title></head><body>
   <h2>Object moved to <a href="https://stackoverflow.com/">here</a>.</h2>
   </body></html>
   ```

In implementing your crawler, you may find java's URL class (http://docs.oracle.com/javase/8/docs/api/java/net/URL.html) a good place to begin.

Notes on editors for Java development: You can choose whatever editors you like for Java development. Options include:

- Simple text editors: You may use any text editor in the VM (`vi` and `nano` are available in the VM) to edit text files. Instead, you may use your favorite text editor from your host OS (e.g., notepad or TextEdit) and transfer the edited file to the VM through the shared directory. Remember that the directory that you specified as the shared directory from the host (e.g., `C:\vm-shared` from Windows) is available at `/mnt/sf_vm-shared` in the VM, which is symbolically linked at `$HOME/shared` as well.

- Java IDE: IDE (integrated development environment) provides a very powerful and convenient programming environment, with features such as constant compile error checking, automatic compiling, and integrated debugging. While we *do not support* a particular Java IDE, many students have reported that Eclipse was particularly easy to use and powerful.

Task 2: Index Wikipedia Dataset Using Elasticsearch

In this task, you will learn how to use Elasticsearch, a popular open-source text search engine, to index a dataset and perform searches.

REST API and `curl` command

Elasticsearch provides a very comprehensive and powerful REST API to interact with your server. For example, you can do:

- Check your server, and index health, status, and statistics
- Administer your server and index data and metadata
- Perform CRUD (Create, Read, Update, and Delete) and search operations against your indexes

For instance, the following HTTP request:

```
GET /_cat/indices?v
```

returns all existing indices in the server.

We can issue an HTTP command to Elasticsearch by using the `curl` command. `curl` is a tool that makes it easy to send an HTTP command to a server and obtain the response from it. For example, if we want to send the above GET command to the Elasticsearch server running in the local virtual machine (localhost) at port 9200, we need to run the following command:

```
curl -XGET 'localhost:9200/_cat/indices?v'
```

At this point, the local Elasticsearch is likely to return a response like:

```
health status index uuid pri rep docs.count docs.deleted store.size pri.store.size
```

which simply means we have no indices yet in the server.

Learn the Basics of Elasticsearch

Now read Elasticsearch Reference to learn how to use it. ***At the minimum***, you ***must*** read the following pages in the reference:

1. Basic Concepts: This page explains a few core concepts related to Elasticsearch. Understanding these concepts is essential in learning how to use Elasticsearch.
2. Create an Index through Exploring Your Data: This sequence of pages explain how to create an index, add/update/delete documents to/from the index, and retrieve a document.
3. Searching through Search Lite: This sequence of pages explain how to perform a search on indexed documents.

Task 2A: Index Using Default Setting

Now that you have learned the basics of Elasticsearch, you need to build our first index using our dataset.

Wikipedia DataSet

Our project1.zip file includes a wikipedia dataset at `data/simplewiki-abstract.json`. This dataset is "succinct summaries" of all pages in the Simple English Wikipedia in the JSON format. More precisely, each line in the file corresponds to one wikipage in the Simple English Wikipedia. Each JSON object includes four "fields", title, abstract, URL, and sections, that corresponds to the title, a short summary, the URL, and the section titles of the page, respectively. Open the JSON file using your favorite text editor and explore the dataset to get familiar with the data. (Warning: this dataset is quite big (~100MB), so it may take a while to open it with a text editor. You may want to take a small subset of the data, say the first 1000 lines, and use the smaller file during exploration and initial development.)

Very First Index Construction

Now, build an index named **task2a** using our Wikipedia dataset. Since we have a large number of documents to index, you may find the bulk API from Elasticsearch useful. To use the bulk API, you may have to "preprocess" the wikipedia JSON file, so that each line of wikipage is preceeded by an "index" command and the appropriate document id. For this preprocessing, you may find the Java skeleton code, `ParseJSON.java` in the project1.zip, useful. The program includes the basic code to read a file with JSON data and parse it. Your job is to fill in your own code to generate an output file suitable for Elasticsearch bulk API. (Note: If you prefer, you are welcome to use another language in parsing the JSON file **as long as it is preinstalled in our VM**. Our VM has python and Node.JS preinstalled, so you can use either python or javascript for the JSON file preprocessing.)

In building your first index, make sure that

1. name of the index is **task2a** (all lowercase) and the type of the documens are **wikipage** and
2. the **id** field of each indexed wikipedia document is set to the **line number of the page** (that is, the wikipage in the first line should get an id=1, second page gets id=2, and so on.)

If your index is built correctly, the index should return the following documents for the given IDs.

id	title	URL
1	April	https://simple.wikipedia.org/wiki/April
34264	1.22.03.Acoustic	https://simple.wikipedia.org/wiki/1.22.03.Acoustic
67384	Ken Starr	https://simple.wikipedia.org/wiki/Ken_Starr
91877	Marking knife	https://simple.wikipedia.org/wiki/Marking_knife
128612	Tidal viloet	https://simple.wikipedia.org/wiki/Tidal_viloet

For example, if you send a command like:

```
curl -s -XGET "localhost:9200/task2a/wikipage/1?pretty"
```

You should get a result like

```
    {
      "_index" : "task2a",
      "_type" : "wikipage",
      "_id" : "1",
      "_version" : 1,
      "found" : true,
      "_source" : {
        "abstract" : "April is the 4th month of the year, and comes between March and May. It is one of four months to have 30 days.",
        "title" : "April",
        "url" : "https://simple.wikipedia.org/wiki/April",
        "sections" : [
          "The Month",
          "April in poetry",
          "Events in April",
          "Fixed Events",
          "Moveable Events",
          "Selection of Historical Events",
          "Trivia",
          "References"
        ]
      }
    }
```


Also, your index should return the following number of matching documents for the corresponding queries on the default `_all` field:

Query	# matching docs
information retrieval	502
the matrix	69559
algebra	55
elasticity	9
elizabeth	202
April 5	3487
wrestler	85

For example, if you send a command like:

```
curl -s -XGET "localhost:9200/task2a/_search?q=information%20retrieval&pretty"
```

(Note that the white space between information and retrieval is URL encoded as %20 since the query is part of a URL.)

You should get a result like


```
{
  "took" : 13,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : 502,
    "max_score" : 17.340786,
    "hits" : [
      {
        "_index" : "task1",
        "_type" : "wikipage",
        "_id" : " 111612 ",
        "_score" : 17.340786,
        "_source" : {
          "abstract" : "Information retrieval is a field of Computer science that looks at how non-trivial data can be obtained from a collect
          "title" : "Information retrieval",
          "url" : "https://simple.wikipedia.org/wiki/Information_retrieval",
          "sections" : [
            "Problem description",
            "Different models",
            "First dimension: the mathematical model",
            "Second dimension: the properties of the model"
          ]
        }
      },
  ...
}
```


Once you have successfully built your first Elasticsearch index, add the sequence of commands that (1) converts the simplewiki-abstract.json file into bulkload file (2) creates the task2a index and loads the data using the bulkload API to the `build.sh` script in project1.zip file. Eventually, you will have to submit the `build.sh` file as part of Project 1 submission.

Task 2B: Index Using Whitespace Analyzer

In the rest of Task 2, you will explore some of the basic text processing techniques such as tokenization, stemming, and stopword removal. These techniques are usually applied to each document prior to the indexing it. 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. For example, a simple tokenizer might split the string up into terms wherever it encounters whitespace or punctuation. It is the ability to tokenize and analyze words, that helps elasticsearch perform exceedingly well on natural language data.

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

- **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."
- **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, we may elect to discard them 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.
- **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.

Text processing is done through what is called an `Analyzer` in Elasticsearch. Elasticsearch gives us a number of alternative analyzers that we can use depending on what type of text processing we may need to perform on our dataset. Here is a few in-built analyzers provided by Elasticsearch:

- **Standard Analyzer**: The standard analyzer is the default analyzer used by Elasticsearch. It divides text into terms on word boundaries (as defined by the Unicode Text Segmentation algorithm). It removes most punctuation, lowercases terms (and optionally allows removing stop words which is not turned on by default).
- **Simple Analyzer**: The simple analyzer divides text into terms whenever it encounters a character which is not a letter. It lowercases all terms.
- **Whitespace Analyzer**: The whitespace analyzer divides text into terms whenever it encounters any whitespace character. It does not lowercase terms.
- **Stop Analyzer**: The stop analyzer is like the simple analyzer, but also supports removal of stop words.
- **Keyword Analyzer**: The keyword analyzer is a â€œnoopâ€ analyzer that accepts whatever text it is given and outputs the exact same text as a single term.
- **Pattern Analyzer**: The pattern analyzer uses a regular expression to split the text into terms. It supports lower-casing and stop words.
- **Language Analyzers**: Elasticsearch provides many language-specific analyzers like 'english' or 'french'.

In Task 2B, you need to create your second index on wikipages by using the *whitespace analyzer*, so that the capitalization in the wikipages is preserved. This allows users to obtain different sets of documents depending on the capitalization in their query. To understand how Elasticsearch decides on whether and which Analyzer to use for a document, you need to learn the concept of *mapping* and *data types* in Elasticsearch.

Learn about Mapping, Data Types, and Analyzer

Read the pages from Mapping and Analysis through Complex Core Field Types (**Important!! Do not skip this reading**) to learn the key concepts related to Elasticsearch Analyzer.

These pages also explain how you can specify a particular Analyzer to index your documents document. (Note: the pages assume that your Elasticsearch contain the indices created from the dataset at https://gist.github.com/clintongormley/8579281, which would be the case if you finished earlier reading.)

Now build a new index for Task 2B which uses `whitespace` analyzer. When you build your index for this task, make sure that

1. name of the index is **task2b** (all lowercase) and the type of the documens are **wikipage**
2. the **id** field of each indexed wikipedia document is set to the **line number of the page** and
3. use **`whitespace`** analyzer for **every field** of the document, **including the `_all` field** (as explained in the Search Lite page, `_all` field is automatically generated by Elasticsearch to support simple queries with no field specifier (a.k.a. Search lite))

   Note: Specifying the analyzer for the `_all` field is slightly different from other regular fields. See this page to learn how you can correctly set the analyzer for the _all field without getting any error.

If your index is built correctly, the index should return the following number of matching documents for the given queries:

Query	# matching docs
information retrieval	343
the matrix	62496
algebra	26
elasticity	3
elizabeth	0
April 5	2635
wrestler	21

Note that the query "elizabeth" does not return any matching documents because our query is in lowercase. Since the `whitespace` analyzer does not perform lowercase conversion, Elasicsearch will not consider "Elizabeth" and "elizabeth" as the same tokens.

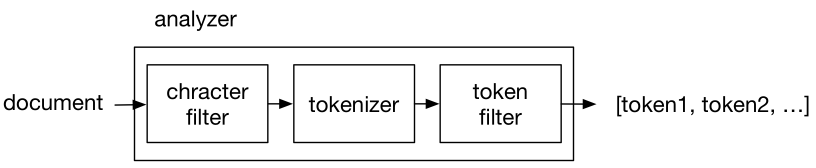
Add the sequence of your Elasticsearch commands to build the second index to the `build.sh` script.

Task 2C: Index Using Custom Analyzer

Sometimes, we may want to specify transformations beyond what are applied by the in-built analyzers. In such cases, we can build our own *custom analyzers* that combine the appropriate character filters, tokenizers and token filters.

Character Filters, Tokenizers, and Token Filters

Any Analyzer in Elasticsearch consists of three transformation subprocesses, called `character filters`, `tokenizers`, and `token filters` as is depicted in the following diagram and as you learned from the Analysis and Analyzers page.



All incoming documents go through a (set of) chracter filter, which transforms individual characterers, then a tokenizer which splits incoming stream of chracters into a sequence of tokens and a (set of) token filter, which transforms the output tokens from the tokenizer.

Read the Custom Analyzer page to learn how you can synthesize your own custom analyzer by combining available filters and tokenizers in Elasticsearch. Once your custom analyzer is created and named, you can use it to analyzer a field of your documents by specifying its name as the analyzer of the field (as part of the `mapping` as you did in Task 2B).

In this task, build a custom analyzer that consists of the following filters and tokenizer:

- Character Filter: HTML Strip Char Filter that strips HTML elements from the text and replaces HTML entities with their decoded value (e.g. replacing & with &).
- Tokenizer: Standard Tokenizer that divides text into terms on word boundaries, as defined by the Unicode Text Segmentation algorithm.
- Token Filters:
  1. ASCII Folding Token Filter that converts alphabetic, numeric, and symbolic Unicode characters which are not in the first 127 ASCII characters (the "Basic Latin" Unicode block) into their ASCII equivalents, if one exists.
  2. Lowercase Token Filter that normalizes all token streams to lowercase.
  3. Stop Token Filter that removes stopwords from token streams. By default, it removes stopwords in `_english_` list, which are "a, an, and, are, as, at, be, but, by, for, if, in, into, is, it, no, not, of, on, or, such, that, the, their, then, there, these, they, this, to, was, will, with".
  4. Snowball Token Filter that stems words using a Snowball-generated stemmer.

  (**Note**: The final output are dependent on the order in which the filters are applied. It is important that you specify the list of token filter **exactly in the above order** to obtain the result that we describe later.)

When you build your index for this task, make sure that

1. name of the index is **task2c** (all lowercase) and the type of the documens are **wikipage**
2. the **id** field of each indexed wikipedia document is set to the **line number of the page** and
3. use a **custom** analyzer for **every field** of the document, **including the `_all` field**. For each filter/tokenizer used for the custom analyzer, use its **default setting**.

If your index is built correctly, the index should return the following number of matching documents for the given queries:

Query	# matching docs
information retrieval	788
the matrix	25
algebra	74
elasticity	23
elizabeth	203
April 5	3487
wrestler	111

Note that in Task 2A, the query "the matrix" returned 62496 hits, while it returns 25 hits here. This is because in Task 2A, all documents that contain either "the" or "matrix" are matched, while here, only the documents with "matrix" are matched due to stopword removal. Also, notice that the query "elasticity" returns more documents here than in Task 2A. This is because of stemming. Documents with the word "elastic" is also considered a match when stemming is applied to documents.

Once you are done, please add the sequence of your Elasticsearch commands to build the new index to the provided `build.sh` script.

Task 3: Construct Evaluation Dataset

Evaluating the accuracy of search results is fundamentally a *subjective* task that requires human judgement. As the last task of Project 1, we ask your help in constructing an evaluation dataset, by which the effectiveness of the search ranking algorithms in later projects will be evaluated. We hope to collect a diverse pool of judgments from all of you, which should allow a reasonable evaluation of various ranking algorithms. Relevance judgments are (query, document) pairs marked as relevant or irrelevant by users. For this purpose, we ask you to come up with 10 queries that could be run on our Wikipedia dataset. You must submit the query, a brief description of the type of results that you expect, and the list of "relevant" pages in a text file named task3.txt. The result pages should be specified by the URL of the pages. For example if your query is "algebra", for which the relevant pages are "algebra" and "linear algebra", it should be formatted as:

```
{ "query": "algebra", "description": "all pages related to algebra", "pages": ["https://simple.wikipedia.org/wiki/Algebra", "https://simple.wi
```

Your Final Submission

Your project must be submitted electronically before the deadline through our CCLE Course Website. Navigate to **Projects** on left of the page, and click on the **Project 1** submission section. If you submit multiple times, we will grade only the latest submission.

What to Submit

The zip file that you submit must be named `project1.zip`, created using a zip compression utility (like using "`zip -r project1.zip *`" command in the VM). You should submit this single file **project1.zip** that has the following packaging structure.

```
project1.zip
|
+- GetWebpage.java
|
+- ParseJSON.java (Only if you use our skeleton code for Wikipedia data parsing)
|
+- build.sh
|
+- task3.txt
|
+- any other file that is needed by your build.sh
|
+- README.txt (Optional)
```

Three key files in your zip file, GetWebpage.java, build.sh, and task3.txt should meet the following requirements:

1. GetWebpage.java: This java file should not use any external Java library. We should be able to compile and run your Java code simply like "javac GetWebpage.java" and "java GetWebpage.java http://www.google.com".
2. build.sh: This file should take the wikipedia datafile located at ./data/simplewiki-abstract.json, preprocess it to make it loadable using batch API, and load the preprocessed file into three indexes, task2a, task2b, and task2c. This entire process can be done simply by executing "./build.sh". That is, if you use our ParseJSON skeleton code, you should compile the Java code as part of build.sh script and run the Java code inside build.sh as well.
3. task3.txt: This file should contain evaluation queries and relevant pages in JSON format.

In your zip file, please **DO NOT INCLUDE** data/simplewiki-abstract.json. This will make your submission file too big. Please ensure that your submission is packaged correctly with all required files. Make sure that each file is correctly named (including its case) and project1.zip contains all files directly, not within a subdirectory. In other words, unzipping project1.zip should produce the files in the same directory as project1.zip. **Significant points may be deducted from your submission if the grader encounters an error due to incorrect packaging, missing files, and failure to follow our exact spec.**

Testing of Your Submission

Grading is a difficult and time-consuming process, and file naming and packaging convention is very important to test your submission without any error. In order to help you ensure the correct packaging of your submission, we have made a "grading script" p1_test available. In essence, the grading script unzips your submission to a temporary directory and executes your files to test whether they are likely to run OK on the grader's machine. Download the grading script and execute it in the VM like:

```
cs246@cs246:~$ ./p1_test.sh project1.zip
```

(if your project1.zip file is not located in the current directory, you need to add the path to the zip file before project1.zip. You may need to use "chmod +x p1_test" if there is a permission error.)

You **MUST** test your submission using the script before your final submission to minimize the chance of an unexpected error during grading. Again, significant points may be deducted if the grader encounters an error during grading. Please ensure that the Elasticsearch service is running before you execute the script. When evertything runs properly, you will see an output similar to the following from the grading script:

```
Compiling GetWebpage.java...
Testing GetWebpage.java...
SUCCESS!

Running build.sh...
.
.
.
Testing Task2A...
SUCCESS!

Testing Task2B...
SUCCESS!

Testing Task2C...
SUCCESS!
```