In this session:

- You will learn how to use the ElasticSearch database
- How to index a set of documents and how to ask simple queries about these documents.
- We will study whether the given texts satisfy Zipf's and Heaps' laws.

1 Running ElasticSearch

ElasticSearch is a NoSQL/document database with the capability of indexing and searching text documents. This database runs as a web service in a machine and can be accessed using a REST web API.

In order to run the service we have to create some configuration files. Given that probably you will have not enough quota in your user space, you can use `/tmp` for the configuration and data files.

The ElasticSearch binaries are in `/opt/elasticsearch-5.4.1/`.

Within the files of the session you have a directory named `ESconf`, follow the instructions in the `Readme` file that you will find inside. After you have completed the instructions you will have ElasticSearch up and running.

To test if ElasticSearch is working you have a script called `elastic_test.py`, you can run it from the command line as:

```
$ python elastic_test.py
```

If ElasticSearch is working you will see an answer from the server or a message indicating that it is not running.

2 Indexing and querying

As mentioned before, ElasticSearch is accessed as a web service using a REST API. This means that we can operate with the DB using the HTTP protocol (just like any web server). ElasticSearch defines a set of methods that correspond to all the actions available. You can access to the full documentation with this link.

To make things simpler we are going to use two python libraries (`elasticsearch` and `elasticsearch-dsl`) to access ElasticSearch. The first one is more general but hides less the complexities of the API calls, the second one is more focused on the search capabilities and is more friendly for sending queries to ElasticSearch. In the scripts that you have both libraries are used depending on the operations performed.

To use a simile with relational databases, an index in ElasticSearch corresponds to a table. There is not a specific schema (set of columns) associated with an index, we can insert different types of documents in an index that can have different fields. We can tell to the DB what fields have to be indexed. All documents sent to the DB are serialized using the `JSON` format, so a document will look like:

```
{"field1": "2017-01-31",
 "field2": "Some text here",
 "field3": 33
}
```

2.1 Anatomy of an indexing

Download these two zipped sets of files: 20_newsgroups and novels. Unzip them in your working directory (or `/tmp` if you do not have disk space enough), to directories 20_newsgroups and novels. They contain respectively:

- Text from 20 usenet groups on various topics, a classic corpus in IR evaluation, from here.
- A number of random novels and other texts in English from the Gutenberg project, with a tendency towards late 19th and early 20th centuries.

You have among the session files a script named `IndexFiles.py`. Open the script with a text editor.

The script has basically two parts, the first one reads all the documents traversing recursively the path that is received as a parameter and creates a list of ElasticSearch operations. There is a method in the API for indexing just one file, but given that we are going to index a lot of them we can use the `bulk` method that allows to execute several ElasticSearch calls as one.

A bulk operation is a special document that indicates the type of operation to execute (`_op_type`), in this case `index`, the index where the operation is performed (`_index`) and the information of the document: the type of the document and the fields of the document. In this case we have used as type `document` and we include two fields in the document `path` for the path of the file and `text` for the content of the file.

The second part of the script operates with ElasticSearch, first we need an `ElasticSearch` object that will manage the connection with the DB, if we have the service in the local machine with the default configuration, no parameters are needed, in any other case we must configure the object adequately. With this object we create the index for the documents (if it already exists it is deleted) and the bulk method is called for performing all the insertions.

Now you can use this script with the documents that you have downloaded, for example:

```
$ python IndexFiles.py --index news --path /path/to/20_newsgroups
```

will insert all the documents from `20_newsgroups` in the index `news`

2.2 Looking for mr goodword

After we have all the documents inside an index we can query ElasticSearch using specific terms or more complicated queries. ElasticSearch has a DSL (Domain Specific Language) for specifying what we want to search. We are not going to enter in the complexities of what can be done, but this kind of DBs allows more flexibility than the classical relational DB. Apart from exact matches we can do fuzzy matches, obtain suggestions, highlight the parts of the document that includes the search term, ... Basically anything that we expect from a web search engine because this kind of DBs are the ones that are behind a query in Google search for example.

The script `SearchIndex.py` allows to query an index in our ElasticSearch DB. It has three flags `--index` is obviously the index and we have also `--text` and `--query` parameter.

The `--text` takes precedence over `--query` (you can not use both) and allows for looking for a word in the `text` field of our documents. It will return all the exact matches with the document id, the path of the file that matches and the highlighted parts that contain the word.

The `--query` flag allows to use the LUCENE syntax for the queries (LUCENE is the open source indexing system all this kind of DB use). This syntax allows boolean operations like AND, OR, NOT (always in uppercase) or fuzzy searches using `~n` with n indicating how many letters of the word can mismatch to consider it a match. Using this flag the query returns the id of the documents, the path and the 10 first characters of the document.

Open the script with a text editor and look to the part of the code where the search query is built. Have a look at the documentation of `elasticsearch-dsl` to understand it better.

Now you can play a little bit with the script, for example execute:

```
$ python SearchIndex.py --index news --text good
$ python SearchIndex.py --index news --query good AND evil
$ python SearchIndex.py --index news --text angle
$ python SearchIndex.py --index news --query angle~2
```

Each search returns the number of documents matched. Observe how this number changes using larger values for n in fuzzy queries.

3 Zipf's and Heaps' laws

Now lets have a look at the word distribution in these texts, and in particular whether Zipf's and Heaps' laws hold. We recommend using the novels corpus because it is much cleaner.

The `CountWords.py` script in the pack reads an index and writes on the standard output all the terms it contains, with their counts. Execute it redirecting the output to a file. You have two flags, the obvious `--index` and an `--alpha` flag that returns the list alphabetically instead of ascending by the number of occurrences of the word. Beware of that the last line is the number of words.

Inspect the file, sort the words lexicographically and remove stupid terms such as numbers, url's, binary or unreadable stuff, dates, etc. Leave only *proper* words. Now we are ready to analyze the data.

For Zipf's law, check if the rank-frequency distribution seems to follow a power law; what are the power law parameters ($f = \frac{c}{(rank+b)^a}$) that seem to describe best what you see?

You can use a spreadsheet for this, trying different triples (a,b,c) to imitate the number of occurrences of the word as a function of the rank. Plotting a graph (either in linear scale or in log-log scale) is also an option. The fitting will involve some trial and error.

You can also use the libraries that python has available for plotting (`matplotlib`, `seaborn`) and function fitting (`numpy`, `scipy`). These are some links that will help you:

- Minimal matplotlib tutorial
- Minimal seaborn tutorial
- Curve fitting with scipy

Do not depend too much on the very first (most frequent) terms, which are noisy. **What matters is the long tail**. Do you approximately get down to frequency 1, 2, 3, ... in the same places with your formula as the data?

Note: It is not enough to just plot the data and check that you get a decreasing curve that tends to zero. Obviously you will get such a curve. The point is: it is truly a power law as these laws predict? (or an exponential? Or something else with another decrease rate?)

For Heaps' law, check if the number of distinct terms in a piece of text with N words contains about $k \times N^{\beta}$ different words for some k and β. So:

1. create indices containing different numbers of novels, or more precisely different quantities of text, as the sizes of the novels are very different.
2. Use the little program to count the total number of words in each index, and the number of different words in each.
3. See if you find k and β that explain your results well.

4 Deliverables

To deliver: Write a short report with your results and thoughts on the Zipf's and Heaps' tests. Explain things like: What best values of (a,b,c) and (k,β) did you find? How did you find them - what method did you use? We are less interested in long lists of words of screen prints than in your conclusions, so please 2-3 pages maximum

You are welcome to add conclusions and thoughts that depart from what we asked you to do. In fact, they'll be highly valued if they are intelligent and show that you can go beyond following instructions literally.

Rules: 1. You can solve the problem alone or with one other person. 2. No plagiarism; don't discuss your work with others, except your teammate if you are solving the problem in two; if in doubt about what is allowed, ask us. 3. If you feel you are spending much more time than the rest of the group, ask us for help. Questions can be asked either in person or by email, and you'll never be penalized by asking questions, no matter how stupid they look in retrospect.

To deliver: You must deliver a brief report describing your results and the main difficulties/choices you had while implementing this lab session's work. You also have to hand in the source code of your implementations.

Procedure: Submit your work through the raco platform as a single zipped file.

Deadline: Work must be delivered within **2 weeks** from the lab session you attend. Late submissions risk being penalized or not accepted at all. If you anticipate problems with the deadline, tell me as soon as possible.