132a Assignment 5: Learning Elasticsearch (Due Wednesday, April 10)

Goals: This assignment is intended to help you get familiar with Elasticsearch using our film corpus to experiment with some of the many features the system offers. This will prepare you for using elasticsearch and flask functionality to implement your projects.

(For background, a high level overview of elasticsearch functionality with several case studies is found at https://apiumhub.com/tech-blog-barcelona/elastic-search-advantages-books/)

You will need to install:

Elasticsearch (download from https://www.elastic.co/downloads/elasticsearch)

Note that elasticsearch requires that **java** (8 or later) is installed on your system. If you do not already have it installed, you can get it from the official Oracle distribution or an open-source distribution such as OpenJDK.

For Windows, a Windows download is available at: https://www.java.com/en/download/manual.jsp After installation, you should verify that the JAVA_HOME system variable has been set by typing java at the command prompt. If you get a java help listing, you are okay.

You will need two python modules: **elasticsearch**, **elasticsearch-dsl**

If you have pip installed, you can install these modules using pip:

>pip install elasticsearch

>pip install elasticsearch-dsl

At this point, you should be able to start the elasticsearch service on your laptop (see instructions on elasticsearch site for your operating system). On Windows, go to wherever you installed elasticsearch and run bin\elasticsearch . (You can shut down elasticsearch by killing the process, e.g., using cntl-C in Windows).

To test it is running, go to your browser and type http://localhost:9200/ You should see the health status of your ElasticSearch instance with the version number, the name and more.

At this point, you are ready to create an application. Download the ES_example archive from latte and extract the files.

Go to the ES_example subdirectory.

Run: python index.py (builds an index called sample_film_index)

You can test whether the index was created successfully, by typing the following url in your browser: http://localhost:9200/sample_film_index

Run: python query.py (creates a query user interface using flask)

In your browser, open http://127.0.0.1:5000 in order to query the application.

NOTE: To stop the flask server, you may have to use cntl-break rather than cntl-C. You will need to stop flask before rerunning python query.py after a change to the code. If you run into problems with either the elasticsearch or flask service (e.g., no response to typing their urls), try stopping and restarting them.

Details of the assignment

Your job is to flesh out and improve the bare bones film retrieval application we provide. There is no one "right way" to do this. You will have to think about use cases and choose the elasticsearch features that best fit the needs of the various fields and potential queries. Online elasticsearch and python dsl documentation is available at
https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html and

https://elasticsearch-dsl.readthedocs.io/en/latest/

Your interface should include fields for:

- Text
- Language
- Country
- Director
- Location
- Starring
- Time
- Runtime
- Categories

A query should require that all entered fields other than the text field completely match retrieved documents. That is, a document should only be returned if it matches the contents of *all user-specified* fields.

Within the text field itself, use conjunctive search (all terms must appear in a matching document) *unless* this results in 0 matches. In case of 0 text matches, inform the user and *rerun the search* using "match at least one term" criteria ("disjunctive search").

Things to consider:

- For each field, you will need to make choices about how to preprocess the content for indexing and matching. **Choose appropriate data types and analyzers for each field (don't rely on elasticsearch defaults).** For example, the starring field contains a *list* of names and users might search using first name, last name, both, or several names. Some fields may need positional indexing, some raw, exact match, etc. Fields with a small number of values could be presented in the UI as a pop-up list.
- Queries entered in the **text** field should look for matching terms within both title and text fields.
- The **runtime** field needs to be converted to an integer for indexing, in order to support range queries over the field. Look out for strings like "One hour 23 minutes".
- Text queries should handle explicit phrases indicated by double quotes. For example, the text query:

  crime drama "philip roth"

  should treat "philip roth" as a phrase.

- Depending on your choice of tokenizer, some words may be split at hyphen and apostrophe boundaries. Think about how to handle queries containing words like feature-length, re-enact, and Re'ah Fahu.
- Remember the difference in elasticsearch between "query" and "filter" in terms of their use in ranking and choose appropriately.
- You may want to *boost the ranking of documents* for which query terms appear in the title.
- Create an easy to use UI (e.g., elasticsearch has autocomplete options)
- Think about what feedback to present to the user regarding stopwords (if you decide to use them) and the cause(s) of 0 results.
- You should highlight query terms that appear in document snippets and documents displayed.
- Comment your code liberally!

Extra credit:

Use elasticsearch's "bucket aggregation" functionality to add faceted search to your application. Faceted search is a way to help users navigate through search results using the values of specific fields of documents in the result set. Aggregate the results of a query by the location and category fields, so that a user can browse the results of an initial query based on values of these fields.

Documentation on page 21 of https://media.readthedocs.org/pdf/elasticsearch-dsl/6.2.1/elasticsearch-dsl.pdf .

Testing: Develop and test your system using a small database of ~10 hand tailored test documents before scaling up to your full corpus. Run enough queries to convince yourself that your system is working properly given what you know about the effects of tf, idf, boosting, filtering, etc., on ranking. Make sure your final tests cover the range of possible query combinations available.

Deliverables:

Code: Two main python files, one will create your index. The second will open the flask UI and allow users to query the index. The command line calls should be:

python index.py

python query.py

readme.pdf: Follow the instructions included in Assignment 2 regarding what should be included in the readme.pdf file. Include running instructions, details about the files in the folder, choices made for elasticsearch field mappings, timing, packages used, test query examples, and any other thoughts or concerns you'd like to share.

Data: two corpus files (test_corpus.json, films_corpus.json).

Submission:

1. **Upload all files (Including readme.pdf) into latte as a single zip file named "<lastname>_<first name>_es".**
2. **Upload readme.pdf separately into the latte assignment titled "readme". (This is for convenience reviewing on latte.)**