Stan Schepers (*20163280*) — August 2021 — Information Retrieval - Code: https://github.com/stanschepers/information_retrieval

Report 1: Lucene

Description

Lucene is an open-source barebone search engine. It is wIn this section we will discuss the functionality of Lucene. Most of the information below is based on the documentation of Lucene. [1]

Document Analysis and Indexing

Lucene has several different analyses for indexing, tokenising content in different ways and languages. The standars are `StandardAnalyzer` and `StandardTokenizer`. `StandardAnalyser` is `StandardTokenizer` (that tokenises given content) with lower case filtering and stop word filtering. It is possible to extend this functionality with self-implemended filter or filters from Lucene like phonetic analysis, ...

Indexing in Lucene is handled by an `IndexWriter` that stores terms and document ids in an **inverted index** using indexable **Skip-Lists** instead of B-Trees. [2] The index will be predominantly be stored in a single non-readable file.

In Lucene you write documents consisting of different sort of fields. Fields can be analysed or stored in the index, if required.

Query Processing

Query processing is handled by `QueryParser`.

Different types of queries are possible e.g. to build:

- **Wildcard query**: allows a search using wildcards in it.
- **Multi term query**: allows a search documententents containing a subset of given terms.
- **Phrase query**: allows a search documententents containing multiple given terms in a certain order.
- **Fuzzy query**: allows a search for non exact words (up to 2 edits) using **Levenshtein** algorithm. This could be used

as spell correction. There are also other ways to implement spell corrections in Lucene.
- **Regex query**: allows a search using RegEx.
- **Boolean query**: allows for boolean logic (AND, OR) for the terms in the query.

Document Search and Retrieval

Lucene provides different ranking methods that were seen in class: **vector-space models**, **theory-based probabilistic models** (e.g. *BM-25*) and **language-based models**. The vector-space model of Lucene is also discussed in class. It defaults to tf-idf similarity. It is implemented as different similarities that

Conclusion

To conclude, this is the instructions to use Lucene for indexing and searching:

Indexing

1. Open storage for index
2. Create documents with different sort of field.
3. Writing them using the specified analyser to the index (Don't forget to commit).

Searching

1. Open the index
2. Build th query using the specified analyser
3. Search the index using a specified ranking method.

Lucene vs. Solr

Solr is a web-based [3] search tool. It has more fully implemented features and uses Lucence under the hood.

Hands-on

In this section I will discuss my hands on experience with Lucene.

Dataset

The data set [4] is the Wikipedia data and name from 42785 random persons. It is published on Kaggle by Sameer Mahajan.

PyLucene

Installing PyLucene is on macOS 11 is a mess and almost impossible as the assigment said so. Many hours and coffee were spent trying it to get it work as I tought using PyLucene would benfit as I am more proficient in Python and have nearly no experience programming in Java. After some frustrating hours I found a solution: Docker, of course. —Why didn't I think of this before—. There is a hassle-free container available on DockerHub, `coady/pylucence` [5]. The Dockerfile is included in the repo. All things are implemented in main.py.

With a little tweak, all tests from Apache PyLucene are working (except the one with the `PythonException`) and can be found in the `test3` directory. These tests and the samples were used as documentention further on. To run the test run this in the container.

```
python -m unittest discover -s /opt/project/test3 -t /opt/project/test3 in /opt/project/test3
```

Own Implementation

PyLucene has almost the same functions, variables, ... as the Java version. So I decided to work on more Pythonic wrapper, keeping performance, flexibilty and simplicty in mind. This implementation will feature most of the functionality that I wrote in the description.

PyLuceneIndex

`PyLuceneIndex` implements the indexing and searching as a Python `dict`, so Pythonista's can use it with their eyes closed. It is possible to set a specified analyser that is used for indexing and query building and similarity for searching. The structure that is used is `(key, value)`. The value will be analysed by the given analyser. The key will be returned when querying. This is so that file search could be done and the filename can be turned and the content indexed.

`PyLuceneIndex` is user-friendly as it works out of the box only requires 3 lines to build an index and query it.

```
""" Most simple program """
index = PyLuceneIndex()
index["key"] = "value test foo bar"
print(list(index["value"]) # output : ["key"]
```

```
""" Using a different index path, different analyser and different similarity"""
index = PyLuceneIndex('/database/', analyser=WhitespaceAnalyzer(),
similarity=PythonClassicSimilarity())
```

The structure of the `Document` implemented is `(key, value)`. The `value` will be analysed by the given analyser under public domain.

```
index["foo"] = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras congue."
index["bar"] = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus."
index["foe"] = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi."

# Searching
print(index["morbi"]) # output: ["foe"]
print(index["lorem"]) # output: ["foo", "bar", "foe"]
print(index[("lorem", 2)]) # output: ["foo", "bar"]  take top-2
```

Deleting documents is possible using the key.

```
del index["foo"]
```

The syntax of PyLuceneIndex is simple. However it is possible to customise a lot:

1. Overriding the method `new_store` allows to get different types of index storing (e.g. MMapDirectory)
2. Analyser and similarity can be set.
3. Not only text can be queried, also a Lucene `query` can be used as query.

```
stanalyser = StAnlyser()  # Own Analyser
feeling_lucky_similarity = random.choice([LMSimilarity(), BM25Similarity(),
TFIDFSimilarity()]) # Feeling Lucky
index_path = "/some/place"

index = PyLuceneIndex(index_path, analyser=stanalyser, similarity=feeling_lucky_similarity)
```

> This is a lot due to the simplicity of Lucene.

PyLuceneSession

When adding items to the index or searching the index we don't want to open, close the index and commit our changes every single time. A Pythonic way to prevent this from happing is to implement a `with` statement. This keyword allows us to clean up code after the exectuion of a certain block. This way we can save the storage, writer and searcher. This also allows for good exception handling as the storage will always be closed if when an expection is raised. This functionality is implemented in the class `PyLuceneSession`.

```
index = PyLuceneIndex()

with PyLuceneIndexSession(index):  # Enter session: open writer and searcher
    for i in range(10):
        index[str(i)] = f"value_{i}" # write things to index
# Close session: commits and closing happens here
results = index["value_5"]  # uses writer and searcher
```

```
""" Adding the dataset """
with open('data/people_wiki.csv', 'r') as read_obj:
  data = list(map(tuple, reader(read_obj)))
with PyLuceneIndexSession(index):
    for _, name, text in data[1:]:
      index[name] = text
```

It is also possible to implement functions that would index large dataset from the different files in Python. **Lucene handles the buffer for commiting to the actual index.** This buffer size can be adjusted. However for this demo I have choosen for simplicity and reproductibilty. Also the use of custom Python Directory is discouraged.

```
""" from test_PythonDirectory.py
The Directory Implementation here is for testing purposes only, not meant
as an example of writing one, the implementation here suffers from a lack
of safety when dealing with concurrent modifications as it does away with
the file locking in the default lucene fsdirectory implementation.
"""
```

QueryBuilder

```
index["Stan Schepers"] = "Computer Science Master Student at UAntwerp, born in 1998."
index["Donald Knuth"] = "computer scientist and professor born in 1938. Has won the A.M.
Turing Award. He opened a building at Campus Middelheim at UAntwerp."
index["Ada Lovelace"] = "was an English mathematician and writer, chiefly known for her work
on Charles Babbage's proposed mechanical general-purpose computer, the Analytical Engine."
```

I wanted to implement an own QueryBuilder. However the query builder from Lucene works well and is easy to understand. So I will demonstrate several types of queries that are possible. Tests of these queries can be found in the code.

Boolean Query

```
index["computer AND professor"]  # output: ["Donald Knuth"]
index["(student or professor) AND UAntwerp"]  # ["Knuth", "Stan"]
index["NOT professor"] # output: "Ada and Stan"
```

Fuzzy Query

```
index["computer AND proffesor~"]  # output: ["Donald Knuth"]
index["computer AND professor"]   # output: ["Donald Knuth"]
```

Wildcard Query

```
index["pro*e*or"] # output: "Knuth as he is pro(f)e(ss)or)  # how to you write proffesor or professor
```

Proximity Query

This query searcher for words that are max. k tokens apart from each other.

```
index['"computer UAntwerp"~6']  # ["Stan Schepers"]
index['"computer UAntwerp"~6']  # ["Stan Schepers", "Donald Knuth"]
```

StAnalyser

In PyLucene it is possible to extend the `StandardAnalyser`. To test this I try to implement my own analyser class, `StAnalyser`. It implements a `EmojiFilter` The `EmojiFilter` filters emojis an replace them by a description of the emoji. So socuments can be searchable using the feelings of the emojis.

```
stanalyser = StAnalyser()
index = PyLuceneIndex(analyser=stanalyser)

# Wanted functionality

index["happy person"] = "Hi, I am a very happy person! 🤗 🍾"
index["Neil Amstrong"] = "I am an astronaut and I flew a rocket!"
index["Stan Schepers"] = "Hi, I am a cs student."

print(index["🙂"]) # output: ["happy person"]
print(index["champagne"]) # output ["happy person"]
print(index["🚀"]) # output: ["Neil Amstrong"]
print(index["👨‍🎓"]) # output: ["Stan Schepers"]
```

Unfortunately I did not succeed in implementing this. For PyLucene there is no documentation. Only the examples and tests is provided as This functionality is mentioned on some places [6], [7] and unanswered StackOverflow questions [8]. However all implementations don't work even after updating them to new syntax. Also in comparison to the Java variant with my Python variant it should work but it throws unclear Java errors. This is maybe because of the extra wrapper that `PythonTokenFilter` is. The attempt of implementation is included in the code. It would use library `emoji` to translate the emojis in text. I know that I could cheat and translate my input of the wrapper but that is not fair of course.

Conclusion

To conclude this project I have learned quite a deal about how the analysis, indexing, score models, query building in Lucene by reading the documentation a lot and searching for quite obscure code samples. This als oby implementing a pythonic wrapper for PyLucene and learning the implementation behind things. I stumpled on some large difficulties using PyLucene.

Note that this assignment was made individually.

1. https://lucene.apache.org/core/8_9_0/core/index.html ↩
2. https://stackoverflow.com/questions/2602253/how-does-lucene-index-documents ↩
3. http://www.lucenetutorial.com/lucene-vs-solr.html ↩
4. https://www.kaggle.com/sameersmahajan/people-wikipedia-data ↩
5. (https://hub.docker.com/r/coady/pylucene/Dockerfile) ↩
6. https://programtalk.com/vs2/python/2212/txtorg/textorganizer/filters.py/ ↩
7. https://gist.github.com/alexstorer/3455295 ↩
8. https://stackoverflow.com/questions/29248537/how-can-i-create-my-own-tokenfilter-in-pylucene-inherited-from-pythontokenfilter ↩