Information Retrieval Project 2023-2024

Sander Marinus - s0205151
Maarten Peirsman - s0207151

January 16, 2024

1 Project Link

https://github.com/Maarten11/information-retrieval-project

2 Problem Definition

We made a search engine that allows users to search through over 500.000 recipes and close to 1.500.000 reviews, using different parameters, as inspired by supercook.com. The available parameters are *Included Ingredients*, *Excluded Ingredients*, *Name*, *Rating* and *Duration*.

The purpose of this search engine is to allow users to specify which ingredients they already have, and present them with possible recipes this would allow them to make. We wanted to allow the user to search for parts of the ingredients of the recipe. This allows us to make suggestions based on recipes for which the user has all ingredients already or suggest recipes for which some extra items are needed.

Next, we wanted to extend on the functionalities present in supercook, by allowing the user the specify a range for both rating and duration. The rating range allows the user to set a lower bound for the ratings the returned recipes can have. For example: setting range to 3 will ensure our system only returns recipes with a rating of 3 of higher. The opposite goes for the duration. The duration allows you to specify a maximum duration. As a consequence, the returned recipes will only have a duration that is smaller than the maximum duration specified.

3 System, algorithms and covered topics

3.1 System

We have built a web server that acts as an api endpoint to facilitate the recipe searching. This api can handle a request taking in which ingredients you have at your disposal. It also allows you to specify certain ingredients you wish to exclude. This would benefit users that have certain allergies, in that they could rule out certain recipes without having to look through them first.

3.2 Algorithms

Our solution makes use of the Apache Lucene project. This is an open-source search software that implements most of the widely used Information Retrieval algorithms. One of the algorithms used is the BestMatch25 algorithm.

3.2.1 BestMatch25

The BestMatch25 is a probabilistic model that aims to improve on the basic Binary Independence Model (BIM). It does this by first using a simple scoring for the query terms in the document.

$$RSV_d = \sum_{t \in q} \log \frac{N}{df_t}$$

Next it improves upon this simple scoring by taking into account the term frequency and document length.

$$RSV_d = \sum_{t \in q} \log \left(\frac{N}{df_t} \right) \frac{(k_1 + 1) tf_{td}}{k_1((1 - b) + b \cdot (L_d / L_{ave})) + tf_{td}}$$

Where

- tf_{td}: term frequency in document d
- L_d (L_{ave}): length of document d (average document length in the whole collection)
- k_1: tuning parameter controlling the document term frequency scaling
- b: tuning paramter controlling the scaling by document length

Adapted from the lecture slides

The default BM25Similarity implementation in lucene uses $k_1 = 1.2$ and $b = 0.75$. We tested this to have different values for b to try and better hold into account the amount of matched ingredients compared to the total amount of ingredients. This is because we wanted the program to rank recipes for which all ingredients are given by the user higher than recipes for which some ingredients are missing.

We also experimented with using lucene's `Vector Space Model` algorithm.

3.2.2 VSM in Lucene

$$score(q, d) = \text{coord}(q, d) \cdot \text{queryNorm}(q) \sum_{t \in q} \left(tf_{td} \cdot idf_t^2 \cdot boost_t \cdot \text{norm}(t, d) \right)$$

Where

- coord: the number of terms of the query found in the document
- queryNorm: to compare across queries
- tf: default normalization of term frequency with square root
- *boost*: user specified boost for terms
- norm: normalization

Adapted from the lecture slides

3.3 Topics seen in class

- Edit distance for spelling correction
- Stemming
- Boolean search
- Ranked Retrieval
- Okapi BM25 model
- Vector Space Model

4 Implementation details

For ease of use on different systems, we decided to build our project using a docker image. We are using an underlying search library called lucene, that was written in Java and thus requires some setup to work properly with python. This setup is facilitated by using the docker image, in combination with docker compose.

4.1 compose

```
services:
  app:
    build: ./src
    volumes:
      - "./src:/usr/src/app"
      - "./data/:/usr/src/data"
    ports:
      - "8000:5000"
```

4.2 Docker

```
FROM coady/pylucene

WORKDIR /usr/src/app

COPY ./requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

EXPOSE 5000

CMD [ "python", "./app.py" ]
```

To get the project to work with python, we used a python binding of the Java lucene library to allow python developers to access the features of lucene in python. This binding is called pylucene and is bundled in the Docker image as seen above.

We also briefly experimented with a python wrapper for the pylucene project, named lupyne. However, we quickly noticed that this would limit the functionality of lucene that we could use, so we decided to scrap lupyne and use pylucene.

4.3 Web server

For our web server, we used Flask. This allowed us to easily expose the endpoints of our API. We created an endpoint `/search` that can be accessed via a `POST` request.

4.3.1 Search endpoint

This endpoint accepts a javascript Form, containing the different parameters that can be used for the search query. As can be seen in the code above, the search required there to be at least a name or included ingredients to search. This is also prevented in the UI.

In this endpoint, the parameters mentioned above are extracted from the form, and we check which parameters are present. Next, we use these parameters to query the recipes. Finally, we parse the received hits to a jsonify-able format and return them.

```python
@app.route("/search", methods=["POST"])
def search_recipes():
    data = request.form

    name = data.get("name", "")
    include = data.get("include", "[]")
    if not name and include == "[]":
        return "Provide at least a name or an included ingredient", 400

    include = json.loads(include)
    if len(include) == 1 and not include[0]:
        return "Include has a wrong format", 400
    exclude = json.loads(data.get("exclude", "[]"))
    duration = data.get("duration", None)
    if duration is not None:
        duration = int(duration)
    rating = data.get("rating", None)
    if rating is not None:
        rating = int(rating)

    searcher, reader = search.get_searcher(INDEX_DIR)
    hits = search.query_recipes(
        searcher, name, include, exclude, duration, rating, limit=10)

    results = util.hits_to_json_response(searcher, hits)

    reader.close()
    return results
```

4.4 Indexing

In `/retrieval/index.py`, we handle the entire dataset. Here, we make a document per entry in the dataset and put each attribute in its proper field. These fields allow for the searcher to search the dataset more efficiently.

Our recipes were stored in a `recipes.parquet` file and the reviews were stored in a seperate `reviews.parquet` file.

4.4.1 Parse Parquet files

Since we used parquet files for our dataset, we needed to extract this data before we were able to index the different entries. We used the `pyarrow` package for this. There are utility functions provided to easily extract data from parquet files and return them as a `pandas` Dataframe.

```python
def pq_to_df(path: str, columns: list[str]) -> pd.DataFrame:
    return pq.read_table(path, columns=columns).to_pandas()
```

We wanted to have ratings for the recipes as well, to give the users to filter the recipes on quality as well. Because of this, we needed to join the data from both files and put them into one `Dataframe`, so we could index them together.

This happens in the following function:

```python
recipes = util.pq_to_df(DATA_DIR + "recipes.parquet",
            list(util.RECIPE_COLUMNS.keys()))
ratings = util.pq_to_df(DATA_DIR + "reviews.parquet",
            list(util.RATINGS_COLUMNS.keys())).groupby([util.ID_COLUMN]).mean()

combined = recipes.join(ratings, on=util.ID_COLUMN, how="inner")
index.index_data(combined, util.COLUMNS, INDEX_DIR)
```

Next, we could index the combined data.

4.4.2 Indexing

The indexing happens in `retrieval/index.py`. Here, the `index_data` function accepts a `Dataframe`, a `column mapping` and an `index path`. The dataframe contains the data that you wish to index, and the column mapping is used to figure out what field types should be used. This way, we can use the lucene built-in `IntPoint` field to allow for easy range queries. All textvariables are also stored as strings, so that they are returned when a document is a "hit" for a query.

Both the duration and the ingredients list receive a special treatment here:

4.4.2.1 Duration

The duration is specified in the "Period of Time" time format. Since we want to allow for range queries, using the `IntPoint` field, we transformed all duration fields to their duration in seconds.

```
def pt_time_to_seconds(pt_time: str) -> float:
    return isodate.parse_duration(pt_time).total_seconds()
```

4.4.2.2 Ingredients list

For the ingredients list, we needed to store them as a single string to allow the searcher to match all ingredients. However, we also wanted to be able to unwrap them to a list again later on. For this reason we made the list of ingredients into a string using `", ".join(value)`. Here, `", "` is a string that does not normally occur which allows us to split the string to a list again afterwards.

```python
def index_data(pd_table: pd.DataFrame, column_mapping: dict, index_path: str):
    """
    Generates an index for the provided data
    Note: the data should be stored in a parquet file

    :param path: string to the parquet file
    """
    assert lucene.getVMEnv() or lucene.initVM()
    env = lucene.getVMEnv()
    env.attachCurrentThread()
    print(
        f"Writing data to index at '{index_path}'",
        flush=True
    )

    # NOTE: Use the english analyzer to enable stemming.
    #       This can be usefull for searching the ingredients.
    #       'apples' and 'apple' will both match to 'apple'.
    analyzer = get_analyzer()

    # Directory to store the index
    directory = get_index_dir(index_path)
    config = index.IndexWriterConfig(analyzer)
    config.setSimilarity(get_similarity())
    iwriter = index.IndexWriter(directory, config)

    for row in pd_table.itertuples():
        # Convert to dict
        recipe: dict = row._asdict()
        doc = document.Document()
        for key, value in recipe.items():
            # Skip index
            if key == 'Index':
                continue
            if value is None:
                # TODO: what to do with missing data?
                continue
            elif key == "Images":
                if len(value):
                    doc.add(document.Field(
                        key, list(value)[0], document.TextField.TYPE_STORED
                    ))
            elif column_mapping[key] == "list":
                new_value = ", ".join(value)
                doc.add(document.Field(
                        key, new_value, document.TextField.TYPE_STORED))
            elif column_mapping[key] == "int":
                # NOTE: unused
                doc.add(document.IntPoint(key, int(value)))
                doc.add(document.Field(
                    key, value, document.StringField.TYPE_STORED))
            elif column_mapping[key] == "datetime":
                new_value = int(pt_time_to_seconds(value))
                doc.add(document.IntPoint(key, new_value))
                doc.add(document.Field(key, new_value,
                        document.StringField.TYPE_STORED))
            elif column_mapping[key] == "float":
                if np.isnan(value):
                    continue
                v = math.floor(value + 0.5)
                doc.add(document.IntPoint(key, v))
                doc.add(document.Field(key, v,
                        document.StringField.TYPE_STORED))
            else:
                doc.add(document.Field(
                    key, value, document.TextField.TYPE_STORED))
        iwriter.addDocument(doc)

    print(f"added to index", flush=True)
    iwriter.close()
    directory.close()
```

4.5 Search

Searching for relevant documents (recipes) is handled by lucene. According to the lucene documentation for the TFIDFSimilarity class, documents are first 'approved' based on a Bolean model. The approved documents are then scored by default using the BM25 algorithm. To approve documents, a query must be constructed. These queries specify how the documents are searched and the terms matched.

We allow searching recipes based on following parameters:

- *name*: (Part of) The name of the recipe.
- *include*: A list of ingredients that the recipe uses.
- *exclude*: A list of ingredients that the recipe may not use.
- *duration*: The maximum duration to create the recipe in seconds.
- *rating*: The minimum average rating given to the recipe by users.

The recipe name is searched using a phrase query. This is because a name somewhat represents a phrase. There is some seperation allowed between the terms in the name. This allows matching recipes that do not completely follow the name term order.

All included ingredients are combined into a boolean query with the OR operator. Using OR allows matching of recipes that do not contain all ingredients. This is allowed because we do not necessarily want to use all ingredients.

Similarly to the included ingredients, all excluded ingredients are combined in a boolean query with the OR operator. This combined query is then negated using a NOT operator (excluded). Combining the excluded ingredients with OR also allows matching on a single ingredient in the list. We don't just want to remove recipes containing all excluded ingredients, but also recipes containing any of the excluded ingredients.

Duration and rating are range queries. These are only used to filter the results and not for scoring.

All of these queries are then combined together. Both the name and the included ingredients queries should be included, while the excluded ingredients are excluded. As mentioned above, duration and rating simply function as an extra filter.

To allow some spelling mistakes, fuzzy matching is used for the terms in the name and included ingredients queries. This matching is done using the Levenshtein distance (lucene documentation). The default allowed distance is 2. To decrease the chance of wrong matches because of this fuzzy matching, the allowed distance of smaller terms is decreased. This is because there is less chance of a spelling mistake happening in one of these terms and there might be more terms 'close by'.

As mentioned in other parts of the report, we decided to use the Vector Space Model (ClassicSimilarity in lucene) for scoring the results. This seemed to give more desirable results according to our criteria. This choice is further explained in the Results section.

5 Evaluation criteria to evaluate performance

To evaluate the performance of our implementation, we chose to use a qualitative evaluation approach. The evaluation is done by carefully constructing some 'query' statements which where executed on our search engine. The results of these queries where then evaluated by checking whether they satisfied our expectations and the limits of the query. For this we used some extra self generated data together with the dataset. This extra data makes that we are sure that there are recipes for certain queries.

This manual evaluation approach was chosen because of two reasons. Firstly, the indexing and searching is done by lucene. The quality of the retrieval is probably tested quite well. The only real impact we have is changing our queries and changing the way documents are scored. These changes

are quite easy to test manually. Secondly, our dataset does not have any ground truth labels. Using such labels for our project would not make a lot of sense. The results of a query can simply be checked by looking at the ingredients specified in the query and those of the results. Using other datasets for testing would also not really make sense, as these datasets may have completely different characteristics.

Thus for evaluating the results of our implementation, we looked at the results of the chosen queries and checked whether they matched our intentions. For ingredients, this is as simple as checking whether the specified ingredients are present in the results. We also decided that recipes with less extra ingredients should be ranked better than those with a bunch of extra ingredients. This follows our intended use case of looking for recipes that use the ingredients that you have available. The other parameters should simply be satisfied by the results. For example, if we want to exclude a certain ingredient, none of the results should contain this ingredient. The same thus follows for specifying a minimum rating or a maximum duration.

6 Quantitative results from the evaluation

The `src/test.py` script runs our chosen queries on the extended dataset. The results of each query are printed to the console. A limit of 15 results is set for the queries, so that there are enough results to evaluate the query, but not too many so that checking them is still manageable.

We first tested these queries with the default `BM25Similarity` implementation. While this gave recipes that satisfied the queries, we noticed that the results where not always ranked as we would want. When searching for recipes with certain ingredients, a lot of the results had a lot of extra ingredients which were not specified. It was difficult to find recipes using only the ingredients that were specified. Changing to the Vector Space Model for scoring gave better results in this regard. Searching for recipes with certain ingredients now manly results in recipes that only use these given ingredients. Results with extra ingredients mainly happen when the search space becomes narrower by for example also searching for recipes with a certain name.

Querying for a certain duration, rating or excluding ingredients also seems to work. All results for these queries satisfy the constraints.

There are however some irelevant results when searchign for recipes with a certain name or with a certain ingredient. When searching for recipes with 'glass milk' in their name, some of the results do not contain any of these terms. They do however contain a term that is similar to milk or glass. This happens because we use fuzzy matching on the terms to allow for spelling mistakes. If a document contains a term that is similar to the searched term, it may thus be returned as a result. This can also be seen when searching for recipes with 'tomato' as ingredient. A lot of the results contain a spelling variant of 'tomato' and of 'potato'. This happens because 'potato' is similar enough to 'tomato'. It is thus counted when scoring the recipe and might get a higher score than a recipe which does not contain 'potato' but does contain 'tomato'.

7 Limitations

7.1 Ingredient autocompletion

There is no autocompletion to add ingredients. However, be chose the more "Information Retrieval" way of handling this by allowing spelling mistakes. The searcher circumvents this, both by using stemming and levenshtein distance.

7.2 Duration range

Currently our system does not allow users to specify a range for the duration, in the sense that they can currenlty only set a maximum duration. It is possible with the current setup to allow for users to specify a range of in which the duration must fall, let's say between 30 mins and 1hr. However, we figured this would not be a common use case.

8 References

- Information Retrieval Course Notes
- https://lucene.apache.org/core/2_9_4/queryparsersyntax.html
- https://lucene.apache.org/pylucene/features.html
- https://www.kaggle.com/datasets/irkaal/foodcom-recipes-and-reviews?resource=download
- https://hub.docker.com/r/coady/pylucene
- https://vitejs.dev/guide/
- https://tailwindcss.com/docs/guides/vite#vue
- https://www.supercook.com/#/desktop
- https://nlp.stanford.edu/IR-book/html/htmledition/queries-as-vectors-1.html
- https://en.wikipedia.org/wiki/Tf%E2%80%93idf
- https://nlp.stanford.edu/IR-book/pdf/08eval.pdf
- https://stackoverflow.com/a/55477743
- https://coady.github.io/lupyne/examples/
- https://lucene.apache.org/core/9_9_1/core/index.html
- https://www.baeldung.com/lucene-analyzers
- https://lucene.apache.org/core/7_0_1/core/org/apache/lucene/analysis/package-summary.html
- https://stackoverflow.com/questions/64714307/how-to-use-queryparser-for-lucene-range-queries-
- https://pandas.pydata.org/docs/user_guide/merging.html
- https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch
- https://stackoverflow.com/questions/1322732/convert-seconds-to-hh-mm-ss-with-javascript
- https://arrow.apache.org/docs/python/generated/pyarrow.parquet.read_table.html
- https://flask.palletsprojects.com/en/3.0.x/