Tweet Recommender System

Nicolosi Andrea - 834178

Abstract

Users are interested in specific arguments. When they surf on the internet they like to see more information that is related to a topic they like. Similarly, when people use twitter they like to see more tweets related to their topics of interest. So, the aim of the project is to create a simple recommender system for tweets. A registered user can search for specific text inside the tweets collection and also check what the system has selected from him/her as recommended tweets (topics of interest). Precision of recommendation is guaranteed through implicit/explicit feedback (tweets liked, keywords search, etc.) obtained through user behavior monitoring.

1. Framework Used

The project is composed of two parts: a client side, used by the user to navigate tweets' collection (do research, like tweets, and so on) and a server used for tweets crawling and to perform operation on tweets collection.
The client was built using Angular 5 and Clever Theme (an HTML theme based on Angular 5 and Bootstrap).
The server was built using Spring Boot 2.1.1 (based on Java 8).
Tweets, and other information, are stored into a MongoDB database. Connection to database is established using Spring Data JPA API. Tweets are indexed using Apache Lucene 7.5.0 API.

1.1 What is Angular?

AngularJS is a development framework with a particular propensity to support Single Page Application. Angular is a platform that makes it easy to build applications with the web. Angular use the MVC (Model View Controller) pattern to allow separation of tasks in presenting data to the user.

Angular is based on TypeScript, a JavaScript super set that compiles to produce a clean JavaScript output.

1.2 What is Spring Boot?

Spring Boot is a project built on the top of the Spring framework. It provides a simpler and faster way to set up, configure, and run both simple and web-based applications. Notable features of Spring Boot are:

- **Auto-configuration:** It sets up your application based on the surrounding environment, as well as hints what the developers provide. The intelligent auto-configuration attempts to auto-configure your application based on what dependencies you added.
- **Standalone:** Literally, it's completely standalone. Hence, you don’t need to deploy your application to a web server or any special environment. Your only task is to use the run command, and it will start.

1.3 What are MongoDB and Spring Data JPA?

MongoDB is a NoSQL database document oriented. MongoDB stores data in flexible, JSON-like documents. MongoDB provides powerful ways to access and analyze data like such as: ad hoc queries, indexing, and real time aggregation.
Spring Data JPA is part of the larger Spring Data family, makes it easy to implement JPA based repositories. It makes easier to build Spring-powered applications that use data access technologies. As a developer you write your repository interfaces, including custom finder methods, and Spring will provide the implementation automatically.

1.4 What is Apache Lucene?

Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform. Apache Lucene is an open source project available for free download. Lucene offers powerful features through a simple API:

- Scalable High-Performance Indexing
- Powerful, Accurate and Efficient Search Algorithms
- Cross-Platform Solution

2. Functional Description

Tweet Recommender System provides two main features: a full text search algorithm and a recommendation algorithm.

2.1 Full Text Search Algorithm

Users can submit a query to the system, selecting a matching type (exact match or similar match) and a tweet category on which the system will search terms contained into the query. Results can be filtered either explicitly using the selected category or implicitly using user's preferences.

2.2 Recommendation Algorithm

The user's behavior is monitored and relevant data are stored into database. In the main page (dashboard), the system show to the user tweets based on his/her preferences. The user's preferences are constituted of some keywords used by the user during a full text search and on the categories he/she likes. Every time a user likes a tweet whose category isn't inside user's categories liked, it is immediately added. Categories and keywords work as a filter while retrieving recommended tweets.

2.3 Sub-features

Other sub-features implemented for a correct functioning are:

- **Login**: User's login, using username and password, is required. If credential are correct, a client-server session is initialized and user information is saved inside it.
- **Registration**: The user can register to the system choosing a username and a password and selecting one or more tweet he/she likes and one or more category he/she is interested in.

- **Add Like**: Inside the application the user can search for some tweets by using full text search feature or by scrolling the list of recommended tweets. If he/she likes some tweets he/she can click on the like button to add selected tweet to his/her list of liked tweets.

3. Technical Description

The server side is the core of the Tweet Recommender System. The server is structured into three layers:

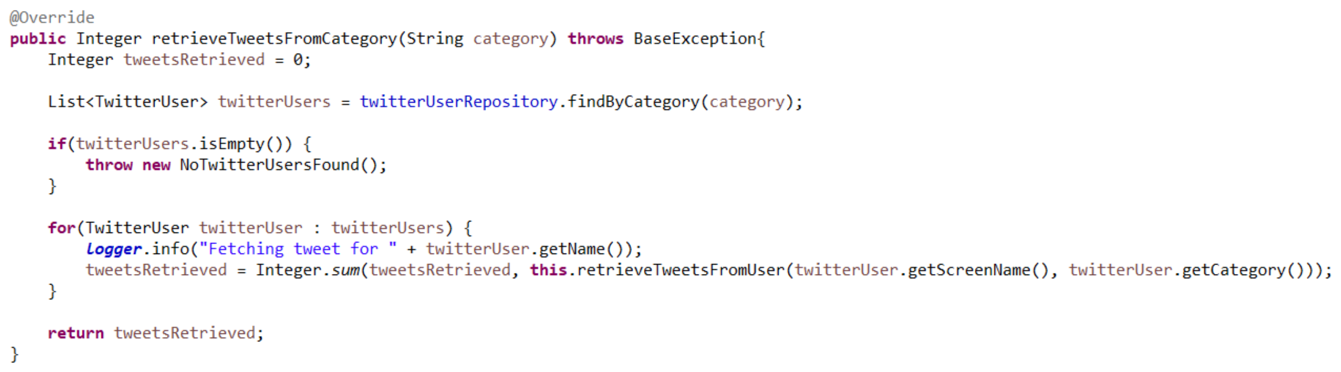
- **Controller Layer**: Which represents the API Rest Controller that manage HTTP requests (GET/POST) performed by the client. It maps JSON request into Java Object and pass them to the Service Layer. There are three controllers: LuceneController, UserController and TweetController.
- **Service Layer**: The middleware between controller and repository. It gathers data from controller, performs validation and business logic, and calls repositories for data manipulation. For each controller there is a corresponding service.
- **Repository Layer**: Layer for interaction with database and performing database operations. There is a Repository for each collection present inside MongoDB.

3.1 Tweets Crawling Process

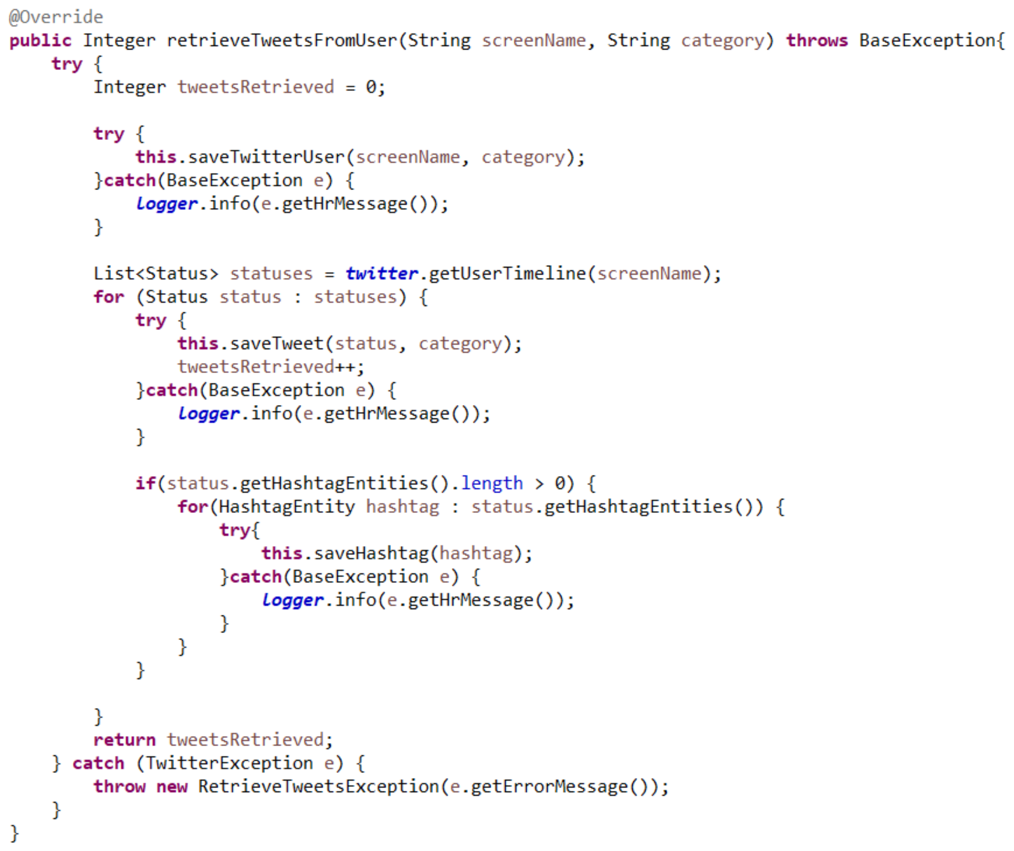
Tweets are retrieved from twitter using Twitter4j API. The server crawling process work as follow:

1. For each category contained into the categories.properties file, twitter users (the people of interest from which we want to retrieve tweets from) are retrieved from database.
2. For each user retrieved, getUserTimeline(screen_name) is called. This API return the last 20 tweets posted by the twitter user.
3. Each tweet returned by the Twitter4j API is saved inside the MongoDB Tweet Collection

Crawling algorithm is showed in the pictures below.



```java
@Override
public Integer retrieveTweetsFromCategory(String category) throws BaseException{
    Integer tweetsRetrieved = 0;

    List<TwitterUser> twitterUsers = twitterUserRepository.findByCategory(category);

    if(twitterUsers.isEmpty()) {
        throw new NoTwitterUsersFound();
    }

    for(TwitterUser twitterUser : twitterUsers) {
        Logger.info("Fetching tweet for " + twitterUser.getName());
        tweetsRetrieved = Integer.sum(tweetsRetrieved, this.retrieveTweetsFromUser(twitterUser.getScreenName(), twitterUser.getCategory()));
    }

    return tweetsRetrieved;
}
```



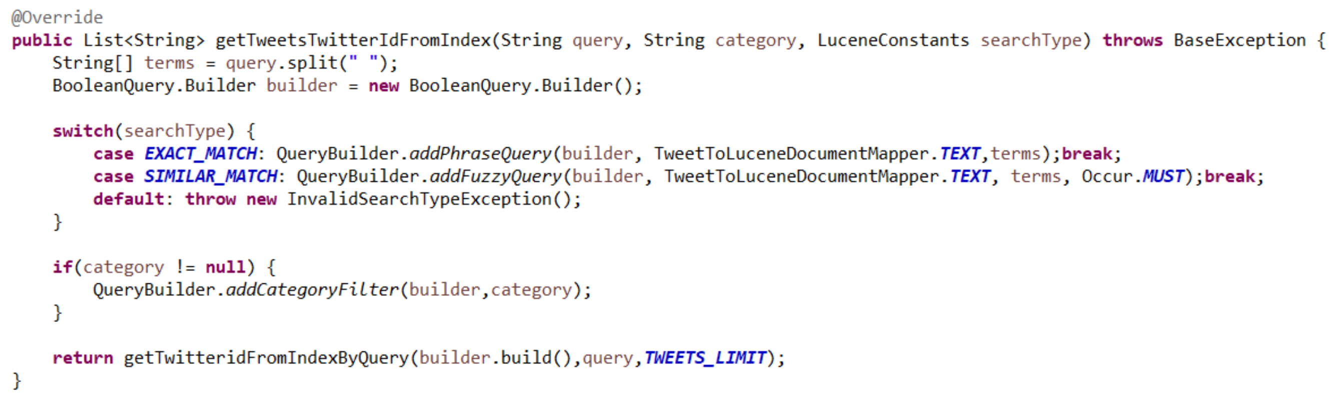
```
@Override
public Integer retrieveTweetsFromUser(String screenName, String category) throws BaseException{
    try {
        Integer tweetsRetrieved = 0;

        try {
            this.saveTwitterUser(screenName, category);
        }catch(BaseException e) {
            logger.info(e.getHrMessage());
        }

        List<Status> statuses = twitter.getUserTimeline(screenName);
        for (Status status : statuses) {
            try {
                this.saveTweet(status, category);
                tweetsRetrieved++;
            }catch(BaseException e) {
                logger.info(e.getHrMessage());
            }

            if(status.getHashtagEntities().length > 0) {
                for(HashtagEntity hashtag : status.getHashtagEntities()) {
                    try{
                        this.saveHashtag(hashtag);
                    }catch(BaseException e) {
                        logger.info(e.getHrMessage());
                    }
                }
            }

        }
        return tweetsRetrieved;
    } catch (TwitterException e) {
        throw new RetrieveTweetsException(e.getErrorMessage());
    }
}
```

3.2 Full Text Search

Full text search is one of the two main features of the system. Given a user query, the algorithm returns the top one hundred tweets filtered by terms inside the query and the category selected. If the "User preferences" flag was selected, the categories liked by the user will be used instead. A "search type" is required to execute the search. Right now, two "search type" are supported:

1. Exact Match: it uses proximity operator to search all tweets that contain all the user's query terms in the same order. Proximity operator is emulated by the use of the PhraseQuery object. For each term inside the query, a Term object is created and added to the PhraseQuery builder. After all terms or the maximum number of terms the system uses are added to the PhraseQuery builder, the PhraseQuery object is added to the BooleanQuery builder with a MUST clause. The MUST clause ensures that all terms must occur inside the tweets.
2. Similar Match: retrieves all tweets which text contains words that are similar to terms inside the query. To do that, the FuzzyQuery object is used. Similarly to the Exact Match, a FuzzyQuery

object is created for each term and then is added to the BooleanQuery builder with a MUST clause to ensure that all terms in the query, or similarly to them, occur inside the tweets



```
@Override
public List<String> getTweetsTwitterIdFromIndex(String query, String category, LuceneConstants searchType) throws BaseException {
    String[] terms = query.split(" ");
    BooleanQuery.Builder builder = new BooleanQuery.Builder();

    switch(searchType) {
        case EXACT_MATCH: QueryBuilder.addPhraseQuery(builder, TweetToLuceneDocumentMapper.TEXT,terms);break;
        case SIMILAR_MATCH: QueryBuilder.addFuzzyQuery(builder, TweetToLuceneDocumentMapper.TEXT, terms, Occur.MUST);break;
        default: throw new InvalidSearchTypeException();
    }

    if(category != null) {
        QueryBuilder.addCategoryFilter(builder,category);
    }

    return getTwitteridFromIndexByQuery(builder.build(),query,TWEETS_LIMIT);
}
```

Tweets are scored by the similarity between terms used by the user and terms inside tweets. Similarity is calculated using the BM25Similarity object. BM25 calculate the similarity between query and document as follows:

$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)}$$

where $f(q_i, D)$ is q_i's term frequency in the document D, |D| is the length of the document D in words, and avgdl is the average document length in the text collection from which documents are drawn. k_1 and b are free parameters, usually chosen, in absence of an advanced optimization, as k_1 in [1.2,2.0] and b = 0.75. IDF(q_i) is the IDF weight of the query term q_i.

3.3 Recommendation

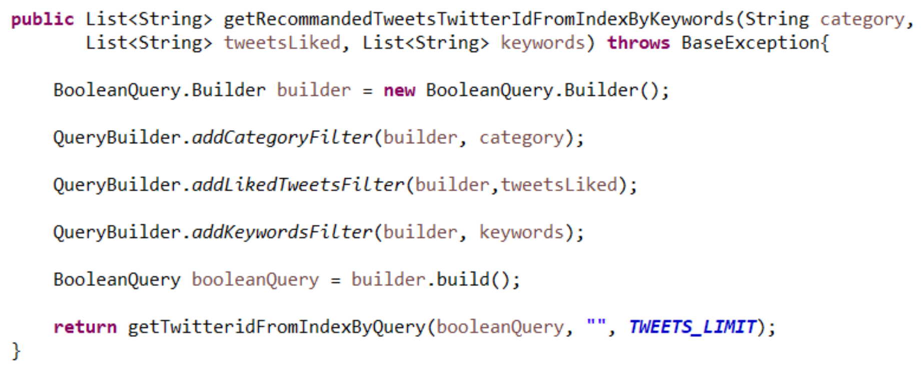
Similarly to full text search, the recommendation algorithm uses Lucene API to perform query inside the index. On the contrary we don't have any query to use to retrieve tweets here. Tweets are filtered using:

1. Categories: For each user's topics of interest the twitterid's retriever function is executed. A TermQuery object is created and added to the BooleanQuery builder with a FILTER clause.

This clause ensures that the tweets' category is the same of the user's topics of interest but does not participate in scoring.

2. Keywords: Keywords used by the user in the previous search are stored inside a list of recent terms, that are cyclically updated on the user's profile. For each term inside the list, a TermQuery object is created and added to the BooleanQuery builder with a SHOULD clause. With this clause terms should appear inside tweets' text. The minimum number of should match is set to one.
3. Tweets liked: A list of tweets liked by the user is stored on the user's profile. This list is used to remove tweets already seen by the user. For each tweet inside the list a TermQuery object is created and added to the BooleanQuery builder with a MUST_NOT clause. This clause ensures that a retrieved tweet is not inside the liked tweets' list.

Like in **3.2** tweets are scored by the similarity between terms used by the user in the previous search and terms inside tweets. Similarity is calculated using the BM25Similarity object.



```java
public List<String> getRecommandedTweetsTwitterIdFromIndexByKeywords(String category,
        List<String> tweetsLiked, List<String> keywords) throws BaseException{

    BooleanQuery.Builder builder = new BooleanQuery.Builder();

    QueryBuilder.addCategoryFilter(builder, category);

    QueryBuilder.addLikedTweetsFilter(builder,tweetsLiked);

    QueryBuilder.addKeywordsFilter(builder, keywords);

    BooleanQuery booleanQuery = builder.build();

    return getTwitteridFromIndexByQuery(booleanQuery, "", TWEETS_LIMIT);
}
```

3.4 Index Creation

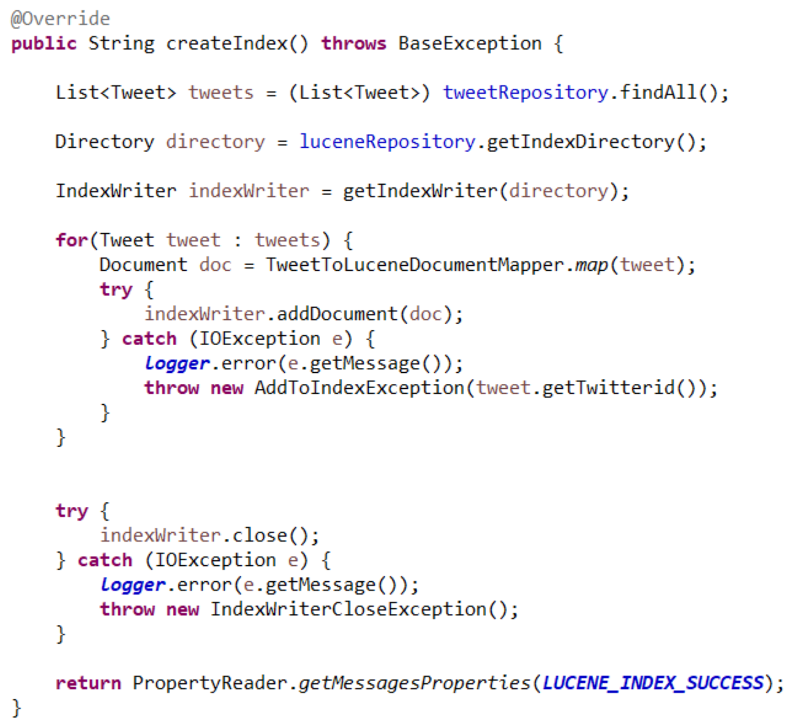
Indexing process retrieves all tweets inside the tweet collection and maps them into a Lucene Document Object. After the mapping phase, each document is then passed to the Lucene Index Writer which writes the documents inside the index. Index is made using the following fileds:

- Text: text is split into terms that are saved inside the index. Saved terms form the index vocabulary.

- Category: tweet's category is saved inside the index to filter tweets using the user topic of interest when a query is performed inside the index.
- TwitterID: twitterid is kept inside the index to retrieve the full tweet's information from database and to remove user liked tweets from query results

There are two reasons why no other information is saved inside the index:

1. More information means that more space is required to save index inside the disk.
2. Fields in a MongoDB document can be indexed so it can be easy to retrieve tweets using their id.



```
@Override
public String createIndex() throws BaseException {

    List<Tweet> tweets = (List<Tweet>) tweetRepository.findAll();

    Directory directory = luceneRepository.getIndexDirectory();

    IndexWriter indexWriter = getIndexWriter(directory);

    for(Tweet tweet : tweets) {
        Document doc = TweetToLuceneDocumentMapper.map(tweet);
        try {
            indexWriter.addDocument(doc);
        } catch (IOException e) {
            logger.error(e.getMessage());
            throw new AddToIndexException(tweet.getTwitterid());
        }
    }


    try {
        indexWriter.close();
    } catch (IOException e) {
        logger.error(e.getMessage());
        throw new IndexWriterCloseException();
    }

    return PropertyReader.getMessagesProperties(LUCENE_INDEX_SUCCESS);
}
```