University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers,
Computer Science and Engineering Department

BACHELOR THESIS

Article Recommender System

Scientific Adviser:
Ing. Mihai Alexandru Ciorobea
Șl.Dr.Ing. Răzvan Deaconescu

Author:
Theodor-Cosmin Didii

Bucharest, 2015

I would like to thank my project supervisor,
Mihai Alexandru Ciorobea, that helped and guided me
through the realisation of this project.

Also, I would like to thank Răzvan Deaconescu
for his unending support in the redaction
of the bachelor thesis.

Abstract

This thesis presents a recommendation system for articles that are rich in content and have certain attributes. This system is needed because an open source tool, able of producing both recommended and related articles, is not accessible. The first step towards choosing an appropriate algorithm for your problem is to decide upon which attributes of your entities you want to focus. Since this is a recommendation system for articles, the main focus is on article content and categorization. Other recommendation systems focus on the content of an article and do not give a great importance to it's date, author, language, or ratings. Also, most of the already existing recommendation systems create and maintain the data in their own database, thus increasing the required storage space.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recommendation systems have become a major research area[3] since the appearance of the first paper on collaborative filtering in the mid 1990s and have since then become popular in both commercial and research communities. The first step toward choosing an appropriate algorithm for your problem is to decide upon which attributes of your entities you want to focus. Since this is a recommendation system for articles the main focus is going to be on article content and categorization. Other recommendation systems focus on the text of an article and do not give a great importance to it's date, author, language, or ratings. Also, most of the already existing recommendation systems create and maintain the data in their own database, thus increasing the required storage space for an application.

In this thesis we describe an open source recommender system that provides an easy to use RESTful API, through Spring Framework's MVC(Model-View-Controller) and grants us access to a multitude of recommendation and related algorithms. These algorithms also use the Stanford Core NLP framework for Natural Language Processing, which helps us improve the quality of our recommendations. Also, a database with fast access is needed to be able to store and process our data in real time. We chose Apache HBase, which fits perfectly for this job, as described in the following chapters.

1.1 Motivation

Adobe is working on a new project that helps content producers like National Geographic and Fast Company bring their content on the web. This project does not have a recommender system. At the moment the recommendations are made by hand. Thus, a specialized recommender system was needed for their articles and specific attributes. Recommender systems that use a part of the attributes already exist, but none of them use all the attributes and are not as easy to extend. Because not all the attributes are used, the recommendations given are not as good.

1.2 Objectives

The main objective is to build a stand-alone application the acts as a Restful API and can offer multiple recommendation options. There should be two types of recommendations:

1. Recommendations based on content (related articles), thus, solving the cold start problem.

   This type of recommendations may use collaborative filtering if specified by the user of the system.

This type of recommendations should make use of all the attributes of an article and give good recommendations.

Each attribute should be given a certain importance in the classifying of the articles.

2. Recommendations based on user history and ratings, giving personalized recommendations for a certain user.

1.3 Related Work

A great amount of research [3] has been carried out on the topic of recommendation systems, both in the academia and in the industry. There is a great deal of diversity in the solutions that currently make up the state of the art in the field, both in regard to algorithms used and the recommendation strategies employed.

In the following sections, we present a couple of the most successful recommendation systems at the moment.

1.3.1 Search Engines (Google, Yahoo, Bing, etc)

These[1] are the most famous types of recommendation systems. They are based on both the content of a site and the particular preferences of an user, determined by their previous search history.

All these systems offer an user the possibility to query the indexed websites by using a certain phrase or combination of words.

The one that offers a tool most similar to an article recommendation system is represented by Advanced Google search[2]. It offers the possibility to find sites that are similar to a web address you already know, but the recommended pages are not that good of a match. By using certain keywords and doing a search based on them, the resulted articles were more alike.

Also, most of the search engines are held by private companies and act like black boxes. You do not have any access to their data and algorithms.

1.3.2 Apache Solr

Solr[3] is an open source enterprise search platform, written in Java, from the Apache Lucene project. It's major features include full-text search, hit highlighting, faceted search, real-time indexing, dynamic clustering, database integration, NoSQL features and rich document handling. Providing distributed search and index replication, Solr is highly scalable and fault tolerant. Solr is the most popular enterprise search engine.

Solr is written in Java and runs as a standalone full-text search server. Solr uses the Lucene Java search library at its core for full-text indexing and search, and has REST-like HTTP/XML and JSON APIs that make it usable from most popular programming languages. Solr's powerful external configuration allows it to be tailored to many types of application without Java coding, and it has a plugin architecture to support more advanced customization.

This solution needs to do its own indexing before any recommendations can be made and it can't be directly integrated with another database. Thus, requiring the duplication of data.

[1] www.google.com
[2] http://www.google.ca/advanced_search
[3] http://lucene.apache.org/solr/

This system does not take into account the previous user search history and does not make personalized recommendations.

1.3.3 Duine Recommender

The Duine recommender[1] is a software module that calculates how interesting information items are for a user. The resulting interest is quantified by a number, called prediction, ranging from -1 (not interesting) to +1 (interesting). When applied in, for example, an electronic TV Guide the Duine recommender can calculate how interesting each TV program is for a particular user. These predictions can be used in various ways: e.g. to provide a user with a list of the top 10 most interesting items, to sort a list of items with the items with the highest prediction at the top, or to present an indication of the interest to the user for each item (e.g. using a number of stars).

The Duine recommender also processes and stores ratings that users give to an information item and interests of the users in aspects of the information (categories, genres, people, topics etc). All data associated with a user is stored in a user profile.

The Duine recommender has learning capabilities. When a user rates an information item, the recommender extracts data from this item (e.g. keywords or genres of a TV program description) to determine the interests of the user. By using smart learning algorithms the recommender slightly adapts the user profile after each rating, based on these interests.

One of the disadvantages of Duine is that it does not offer recommendations based only on content so, it wouldn't be a good fit for the presented problem.Also, it doesn't solve the cold start problem very well, since it's based only on learning and prediction algorithms.

Table 1.1: Solution Comparison

Recommendation System	Collaborative Filtering	Content based	Hybrid	Open Source	Integrates with Existing Database
Search Engines	✓	✓	✓	✗	✗
Apache Solr	✗	✓	✗	✓	✗
Duine	✓	✗	✗	✓	✓
Our Recommender	✓	✓	✓	✓	✓

[1] http://www.duineframework.org/

Chapter 2

Recommendation Algorithms

A great deal has been written about recommendation and related algorithms. In this chapter we describe the algorithms used and we explain our choices.

There are 3 types of recommendation algorithms:

1. Recommendations based on content

   This type of recommendations are solely based on the content of the item to be recommended.

   It uses various text based algorithms in order to calculate the similarity between two items.

2. Recommendations based on collaborative filtering

   This type of recommendations are based on previous user history and ratings.

   The main purpose is to find users with simillar interests to a certain user and recommend items that that are preferred by most of them to that initial user.

3. Hybrid recommendations

   This type of recommendations combine the previous presented.

All 3 types of algorithms can be used, independently or by combining the obtained results. In the following sections we present the implemmented algorithms and why we chose them.

2.1 Algorithms for Related Articles

The related articles recommendation returns a list of articles related to a certain article. We have implemmented both a hybrid and a content based approach. A user may choose to use only one approach or combine them for better results.

2.1.1 Collaborative Filtering

In order to obtain better results, we may use collaborative filtering to do article grouping by user preferences. Usually, users fit into a certain profile and read articles that are related to each other. Using this technique we can reduce the number of articles on which we are going to apply the article similarity algorithm. We use the following iterative algorithm[2] to find articles that are related to article A1:

```
1         For each user U that read A1
2                 For each article A2 read by user U
3                         Save in common article list that a user read
                              both A1 and A2
4         For each article A2 in common article list
5                 Compute the similarity between A1 and A2
6                 Save the computed similarity in related article list
7         Sort the related article list by similarity in descending
             order
8         Return the related article list
```

Listing 2.1: Item to item collaborative filtering

2.1.2 Computing Article Similarity

In order to obtain the best related article list we have to take advantage of all the attributes of the articles at our disposal. We compute the similarity using the:

- **Date created**: used to chose between two articles that have the same related score with the current article
- **Title**: find the similarities in the titles of two articles by using natural language processing
- **Short title**: find the similarities in the short titles of two articles by using natural language processing
- **Department**: check if the articles belong to the same department by comparing the strings
- **Category**: check if the articles belong to the same category by comparing the strings
- **Importance**: articles with the same importance are shown upper in the related articles
- **Publication**: if the articles belong to the same publication the similarity is going to be greater
- **Language**: the articles should be written in the same language in order to be related
- **Author**: if the articles have the same author the similarity is going to be greater
- **Keywords**: if the articles have more common keywords determined by natural language processing analysis then the similarity is going to be greater
- **Ratings**: if the articles have more readers and a higher mean rating, the similarity is going to be greater
- **Collection Reference**: if the articles belong to the same collections the similarity is going to be greater
- **TFIDF** (Term Frequency Inverse Document Frequency) similarity

Each of these fields has a certain importance in the final resulted similarity between articles.

2.1.2.1 Natural Language Processing

In order to compute and improve the similarity between two strings we employ the following three methods:

1. Levenshtein Distance

   We compute the number of characters needed to change one string to another and divide this value by the length of the bigger string. We obtain a value between 0 and 1.

   This method is faster than the next one and is better suited for generating related articles on the fly.

2. By comparing character pairs

   We compute the number of common character pairs of the two strings, multiply it by two and divide it by the number of combined character pairs. In the end, we obtain a value between 0 and 1.

   This method gives better results than the previous one.

3. Lemmatization

   In order to obtain better TFIDF results we use the Stanford NLP (Natural Language Processing) framework for lemmatization.

   Lemmatization is the process of doing a vocabulary and morphological analysis on a text in order to reduce the words to their proper base form.

   It is important to not confuse lemmatization with stemming.

   Stemming usually refers to a crude heuristic process that just chops off the end of a word in order to obtain good results, most of the time. This is usually achieved through the removal of the derivation affixes.

   For instance:

   (a) The word "*am*" will be reduced through lemmatization to the base form "*be*". This form can not be deduced through stemming, as it requires a dictionary look-up.

   (b) The word "*meeting*" can either be a noun, in which case the lemma is going to be "*meeting*", or the ing form of the verb "*meet*". This difference can be deduced by the lemmatizer through the understanding of what type of speech the word actually is from the context. Stemming will reduce the word to the form "*meet*", which is not correct in certain contexts.

   (c) The word "*walking*" has the lemma "*walk*", form which can be deduced both through stemming and lemmatization

2.1.2.2 Term Frequency Inverse Document Frequency (TFIDF)

In order to compute the TFIDF similarity we use the following formulas:

1. Computing the term frequency of a term T in a document, D

$$tf(T, D) = \sqrt[2]{\frac{tfreq(T, D)}{tnum(D)}}. \tag{2.1}$$

   Where tfreq (T, D) is the number of occurrences of a term T in the document D and tnum (D) is the total number of terms in D. A tf value depends on the number of term occurrences. This method cannot characterize documents accurately because it is not able to distinguish important words from trivial words sufficiently. In order to obtain a better TFIDF value we also use stemming and lemmatization on the available words.

2. Computing the inverse document frequency of a term T in a document.

$$idf(t, D) = \log \frac{N}{|t \in D|}. \tag{2.2}$$

Where N is the total number of documents in the corpus and $|t \in D|$ is the number of documents where the term t appears. If the term is not in the corpus, this will lead to a division-by-zero. It is therefore common to adjust the denominator to $1 + |t \in D|$

3. Computing the TFIDF [7]

$$tfidf(T, D_1) = tf(T, D_1) \times idf(T, D_1) \tag{2.3}$$

4. Computing the TFIDF similarity

We use the Cosine similarity in order to do this.

$$\begin{aligned} CosineSimilarity(D_1, D_2) = \cos(\theta) &= \frac{D_1 \times D_2}{||D_1|| \times ||D_2||} \\ &= \frac{\sum_{i=1}^{n} tfidf(T_i, D_1) \times tfidf(T_i, D_2)}{\sqrt[2]{\sum_{i=1}^{n} tfidf(T_i, D_1)^2} \times \sqrt[2]{\sum_{i=1}^{n} tfidf(T_i, D_2)^2}} \end{aligned} \tag{2.4}$$

Because the Cosine similarity doesn't take into account the number of common words or the article size, it does not give good results on it's own, so, we use the following formula, where D1 is the document for which we want the related article list:

$$\begin{aligned} similarity(D_1, D_2) = (CosineSimilarity(D_1, D_2)) \times \\ \frac{(NumberOfCommonWords(D_1, D_2))}{max(NumberOfWordsInD_1, NumberOfWordsInD_2)} \times \\ \frac{1}{\sqrt[5]{NumberOfWordsInD_2}} \end{aligned} \tag{2.5}$$

2.2 Algorithms for Recommended Articles

An algorithm used to find users with similar interests and recommend items.

In order to find users with similar interests, we can use the Pearson correlation:

$$P_{a,u} = \frac{\sum_{i=1}^{m} (r_{a,i} - \overline{r_a}) \times (r_{u,i} - \overline{r_u})}{\sqrt[2]{\sum_{i=1}^{m} (r_{a,i} - \overline{r_a})^2 \times \sum_{i=1}^{m} (r_{u,i} - \overline{r_u})^2}} \tag{2.6}$$

Where r $_{a,i}$ is the rating given by user a to item i and $\overline{r_a}$ is the mean rating given by user a to the corated items.

In order to find items that may interest the user, we can use the following formula:

$$p_{a,i} = \overline{r_a} + \frac{\sum_{u=1}^{n} (r_{u,i} - \overline{r_u}) \times P_{a,u}}{\sum_{u=1}^{n} P_{a,u}} \tag{2.7}$$

Where $p_{a,i}$ is the predicted rating of user a for item i.

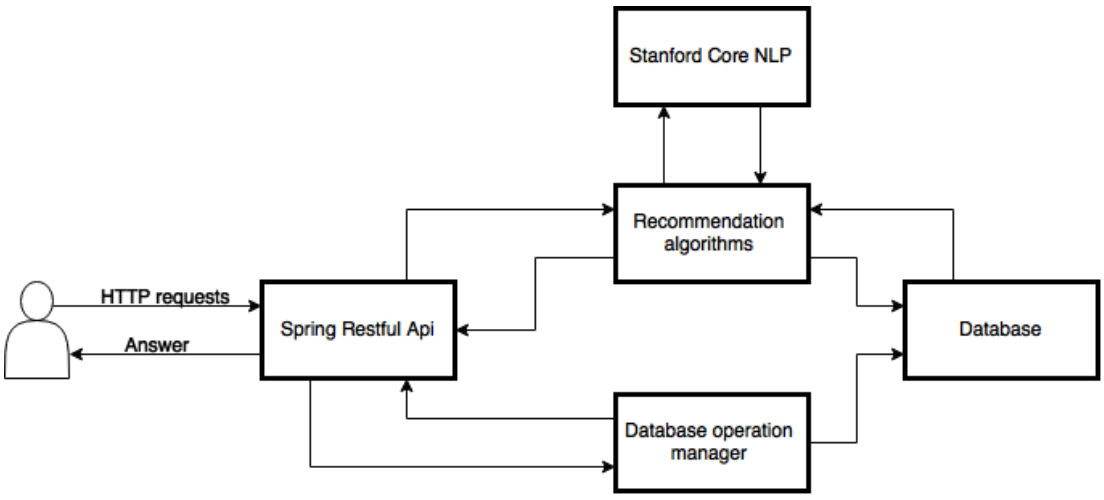
Chapter 3

System Architecture and Workflow

Humans are known for their ability to combine existing elements and create something new. We had a large number of frameworks and solutions to chose from and in the following sections we are going to talk about the various architecture and technology choices. Also, we are going to present the workflow of the system.

3.1 Recommendation System Architecture

Figure 3.1: System Architecture



As we can see in Figure 3.1, the user can interact with our system through a REST API which is provided by Spring Framework's MVC (Model-View-Controller). Based on the controller called by the user, the database operation manager, which uses the databse or the Recommendation algorithm is called, which uses the Stanford Core NLP framework and databse. We use a REST API since it is very easy to use and we needed a NLP framework in order to improve our recommendations. The database is needed for persistence between sessions, fast access to our items and storing our data on the disk, since it is impossible to keep it all in the RAM memory.

3.1.1 Programming languages

There were a multitude of programming languages to chose from. In the following subsections we are going to discuss what languages we chose, why and for what purpose. We were influenced in our choices, by our previous technical experiences, by the ease of use of the presented technologies, the community support and documentation available.

3.1.1.1 Java

Java[1] is a general purpose language that is class based, object oriented and concurrent. It is designed to have as few implemmentation dependencies as possible and be a "write once, run anywhere" type of language, meaning that compiled java code can run on all platforms that support java, without the need of recompilation. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of the computer architecture.

It is a language that supports multiple paradigms, as: object-oriented, structured, imperative, functional, generic, reflective, concurrent. In our implemmentation we use the imperative, object-oriented and concurrent paradigms. By combining the object-oriented and imperative paradigms we managed to obtain a well structured and easy to read code base. By using the concurrent paradigm we are taking full advantage of the current multi-core arhitectures. Another reason for choosing Java is the great community and documentation. At the time of writing this document, Java was the most used language in the industry. Due to this, we can easily find support and documentation on any framework and problem we may stumble upon during the implemmentation of this system. Also, a great variety of frameworks exist for this language.

3.1.1.2 Javascript

Javascript is a high level, dynamic, untyped and interpreted programming language. Alongside CSS and HTML, Javascript is one of the pillars of the current World Wide Web. In the last couple of years it has gained a great popularity in the detriment of action script, which was a programming language for the Adobe Flash Player. This movement towards Javascript is also because most mobile devices do not support Adobe Flash Player. Even though, Javascript may also be used as a backend programming language, since the launch of Node.js, which is an open source, cross-platform runtime environment for server-side and networking applications, the most common usage of Javascript is as a client side scripting language. This means that the Javascript code is sent along with the HTML to the browser, which then interprets it. A great advantage is the fact that the execution of the javascript code is done on the client, thus, in order to use it you just need to serve a file to the client.

In our implemmentation we use Javascript and Node.js to create a demo application which easily integrates with our Recommender System. Javascript was used to create the graphical user interface, since we have an abundance of frameworks that enable us to create a quick and beautiful user interface. Because of various security issues, Javascript is not allowed to make cross domain requests, so we use Node.js in order to create a proxy between our Recommender System and the Javascript client. Node.js is also used to serve the files to the client. Also, in order to better show our Recommender System's capabilities, we built a Chrome plugin that could easily integrate with Adobe's existing platform. Since, Javascript is one of the pillars of the World Wide Web and our Recommender System was meant to easily integrate with a web application, Javascript was the first choice for our demo applications.

[1] https://java.com

3.1.2 Frameworks

A framework is like a black box that offers certain functionalities to it's user and does not require him to have any knowledge of what is going on behind the scenes. The basic principle of a framework is: Not having to reinvent the wheel. By using them we can build faster and better software. Faster, because it allows the programmer to focus on implemmentation specific code, while re-using generic modules, without being tied to the framework itself. Better, because a framework ensures you that the code you are running is in full compliance with the industry's rules. It is structured, and both upgradable and maintanable. In the following sections we describe the frameworks used by our system and why we made these choices.

3.1.2.1 Spring Framework

Spring framework[1] is one of the most popular Java application frameworks. It is an open source platform, which appeared in 2003 and it offers a wide range of modules, like testing, transaction management, Model-view-controller, remote access framework, etc. From all these modules, we use only the Model-view-controller part, in order to create the RESTful API of our system. The only difference between a traditional MVC controller and a RESTful controller is the fact that in place of the server rendering the result on a view, it will just write an object's data, directly to a HTTP response, in a JSON format.

```
1 @RestController
2 public class UserAddController {
3
4     @RequestMapping(method = RequestMethod.POST, value = "/useradd")
5     public void userAdd(@RequestParam(value = "userId", defaultValue
          = "") String userId) {
6         // execute actions
7     }
8 }
```

Listing 3.1: Spring framework REST controller code

As we can see in the above Listing 3.1, adding a controller is easy and we can set the request method and required parameters. We can also return a model, in JSON format, if needed. The only requirement is that the object returned has getters for each of it's attributes. Due to the easiness of use, great popularity and the fact that it enforces some of the java principles, by requiring us to set accessors to the used classes, we can say that Spring is the best choice for our system.

3.1.2.2 Apache HBase

Hbase[2] is an open source, non relational and distributed databse, modeled after Google's Bigtable[5]. It is written in Java, which is a great advantage, since it is really easy to integrate with our existing code. Unlike realational databases, the structure of a non relational database is very simillar to that of a hash map. This grants us an easier control over the content of the database and faster access time. Most of the operations can be done in constant time. HBase also has an API that supports multiple types of queries, simillar in functionality to those of a SQL database. Since HBase is a distributed database, it also obeys the CAP theorem, also known as Brewer's theorem, which states that it is impossible for a distributed system to provide all of the following:

[1] http://projects.spring.io/spring-framework/
[2] http://hbase.apache.org/

- Consistency - The same data is seen by all nodes at the same time.
- Availability - If a number of nodes from your cluster fail, then the system will still be available.
- Partition tolerance - The system will continue to operate, even if the cluster has been split into multiple partitions, that are unable to exchange messages, due to network failures.

Hbase has appeared due to the need of processing massive amounts of data for the purpose of language processing and is a CP type system. Since, we needed a fast and reliable database, that could process the large amounts of text data that we obtained from our articles and which could easily be integrated with Java, HBase was the right choice for our recommender system. Another reason for choosing HBase was the fact that it has a great community and support.

3.1.2.3 Stanford CoreNLP

Stanford CoreNLP[4] provides us with a set of NLP processing tools. Even though, this is a powerful tool, capable to give the base forms of words, their parts of speech, whether they are names of companies, people, etc., normalize dates, times, and numeric quantities, and mark up the structure of sentences in terms of phrases and word dependencies, indicate which noun phrases refer to the same entities, indicate sentiment, etc we use only a small part of it's capabilities. Stanford CoreNLP is an integrated framework which can be used with ease and we can choose what tools to enable or disable by simply setting the options given to the framework. From all the existing options we use only the "tokenize", "ssplit", "pos" and "lemma", which we describe below.

The purpose of the "tokenize" option is to tokenize the text. We need this in order to split large chunks of text into separate words, thus being able to lemmatize each word and use it to get better similarity between articles. The purpose of the "ssplit" property is that of splitting the text into sentences for the future analysis of the meaning of each word based on the context of the sentence in order to reduce it to a basic form by using the lemma property. The purpose of the "pos" option is to enable the Part-Of-Speech Tagger (POS Tagger) of the Stanford CoreNLP. A POS Tagger is a tool that reads text in a language and assigns parts of speech to each word, such as noun, verb, adjective, etc.. We need it in order to determine what part of speach each word is and based on that, we can reduce the word to a simpler form, by using the "lemma" property. The purpose of the "lemma" property is that of applying the lemmatization process to all the tokens of the document. Lemmatization is the process of grouping different forms of a word so they can be processed as a single item. In most of the languages, words are used in several inflicted forms. For example the verb "to be" may appear as "was", "am", "being", etc. The base form of the word, in this case, would be the one you can find in the dictionary, "be" and this is called the lemma of the word. If we combine the part of speech, determined from the context, with the base form of the word, we get something called the lexeme of the word.

Another process, closely related to lemmatisation is called stemming. The main difference is that in the stemming process we do not take into account the context of a word, but only the word itself, and therefore we ca not make a difference between words, based on the part of speech. For example, the word "beings", will always be reduced by a stemmer to the base form "be", which is wrong "All beings are like each other in existing". The lemmatisation process will determine, based on the context, that the word is a noun and has the basic form "being". A stemmer will be able to reduce the word "walking" to the correct form "walk", just like a lemmatizer.

If the application does not need great accuracy, then a stemmer is recommended, since the running time is much faster and the implemmentation is simpler. We chose a lemmatizer in order to increase the quality of our similarity results. The lemmatizer is used only in the article adding

stage, when creating the TFIDF entries, in order to be able to make fast recommendations. If we need to compute word similarity, during the recommendation stage, then we are using the Levenshtein Distance or a simple word similarity algorithm, based on pairs of letters of the word, as presented in the Recommendation Algorithms chapter.

```
1  // Create StanfordCoreNLP object properties, with POS tagging
2  // (required for lemmatization), and lemmatization
3  Properties props;
4  props = new Properties();
5  props.put("annotators", "tokenize, ssplit, pos, lemma");
6
7  // StanfordCoreNLP loads a lot of models, so you probably
8  // only want to do this once per execution
9  StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
10 // Create an empty Annotation just with the given text
11 // and convert the text to lower case.
12 Annotation document = new Annotation(documentText.toLowerCase());
13 List<String> lemmas = new LinkedList<String>();
14 // run all Annotators on this text
15 pipeline.annotate(document);
16
17 // Iterate over all of the sentences found
18 List<CoreMap> sentences = document.get(CoreAnnotations.
       SentencesAnnotation.class);
19 for (CoreMap sentence : sentences) {
20     // Iterate over all the tokens in a sentence
21     for (CoreLabel token : sentence.get(CoreAnnotations.
           TokensAnnotation.class)) {
22         // Retrieve and add the lemma for each word into the
23         // list of lemmas
24         lemmas.add(
25                 token.get(CoreAnnotations.LemmaAnnotation.class));
26     }
27 }
28 return lemmas;
```

Listing 3.2: Stanford Core NLP lemmatizer code

3.1.2.4 Apache Maven

Apache maven[1] is a buid and project management system used primarily with Java. It's main functionalities are that of building a project and defining it's dependencies. A XML files is used to define the project's dependencies, repos from where they can be downloaded, the order in which the build executes and the required directors and plugins. In order to be able to easily setup our system on different platforms, we needed a dependency system. Since Apache Maven is a popular system, compatible with Java, that has quite a long history and a great community it was our choice.

[1] https://maven.apache.org/

3.2 Recommendation System Workflow

There are multiple actions that you can request from the recommendation system through the Spring RESTful API. In the following sections we describe the implemented use cases. All the data is returned as a JSON object.

3.2.1 Add user

A PUT request is made to the RESTful API with the required data for adding a friend of an user. The minimum data required for an user is the user id and friend id. The request is intercepted by the API and the new friend is added to the user's top friends list. If the operation is successful then the user receives "true", otherwise the user is informed of the error.

3.2.2 Add user friend

A PUT request is made to the RESTful API with the required data for adding a friend of an user. The minimum data required for a user is the user id and friend id. The request is intercepted by the API and the new friend is added to the user's top friends list. If the operation is successful then the user receives "true", otherwise the user is informed of the error.

3.2.3 Add user direct recommendation

A PUT request is made to the RESTful API with the required data for adding a direct user recommendation. These are supposed to be recommendations made manually by a user. The minimum data required is the user id and an item id. The request is intercepted by the API and the user entry is fetched from the database and the new item is added to the items directly recommended list of the user. If the operation is successful then the user receives "true",otherwise the user is informed of the error.

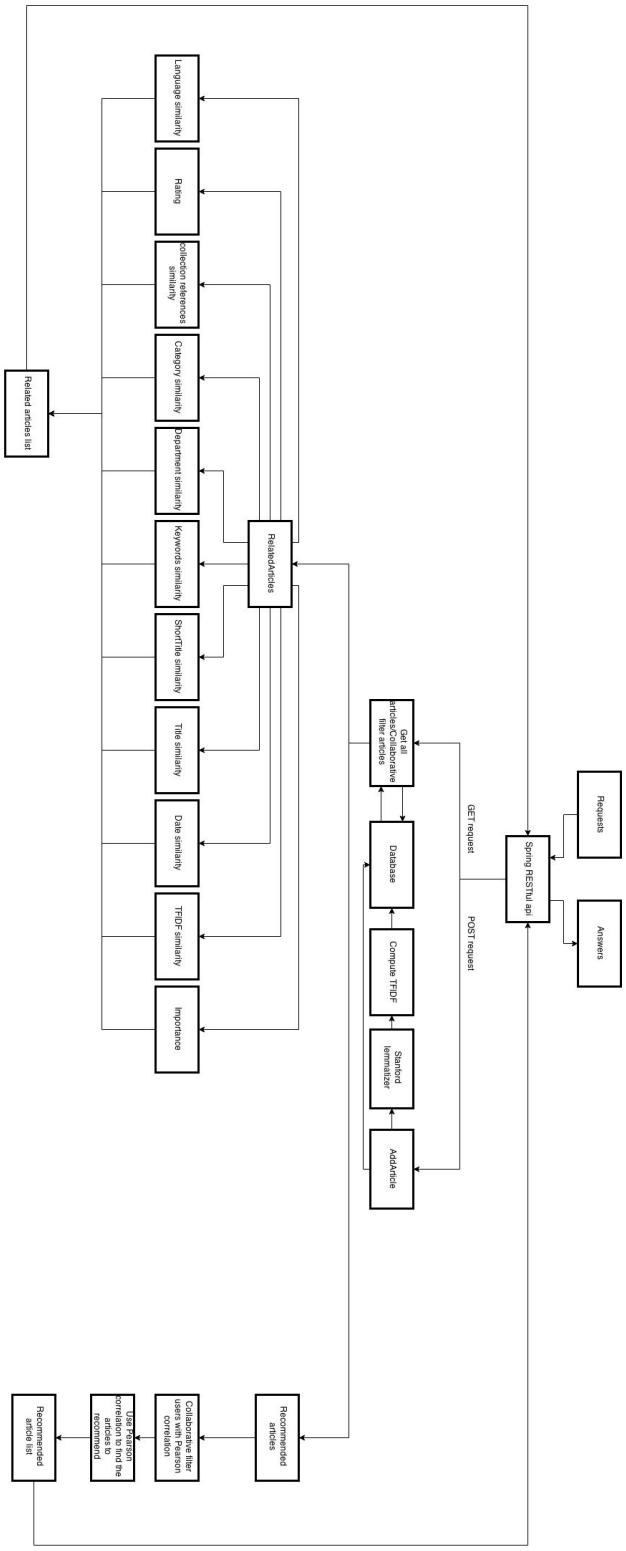
3.2.4 Delete user

A DELETE request is made to the RESTful API with the required data for deleting an user. The minimum required data is the user id The request is intercepted by the API and the database entry is deleted. If the operation is successful then the user receives "true", otherwise the user is informed of the error.

3.2.5 Update user article history

A PUT request is made to the RESTful API with the required data for updating an user's article history. The read article history can be updated and a rating of a certain article can be added. The minimum required fields are the article id and the user id. The request is intercepted by the API and the user entry is fetched from the database. The new item is added to the article history list of the user. If the operation is successful then the API returns "true", otherwise the user is informed of the error.

Figure 3.2: System Workflow



3.2.6 Article recommendations based on friends direct recommendations

A GET request is made to the RESTful API with the user id. The request is intercepted by the API and all the directly recommended articles of the top friends of the user are returned. The top friends list can be added manually through the API. If a top friends list does not exist, one will be generated by using the user's previous article history. If the operation is successful then the user receives "true", otherwise the user is informed of the error.

3.2.7 Add article

A PUT request is made to the RESTful API with the required data for adding a new article. The request is intercepted by the API and the new article is added to the database. The minimum required data is the article id. If the operation is successful then the user receives "true", otherwise the user is informed of the error.

3.2.8 Get articles

A GET request is made to the RESTful API in order to get the articles from the databse. The user may specify how many articles he wishes to receive or he may request a specific article. The articles are fetched from the database and returned to the user. If the operation is successful then the user receives the article list, otherwise the user is informed of the error

3.2.9 Get recommended articles

A GET request is made to the RESTful API in order to get the recommended articles for a user The minimum required data is the user id for whom we need the recommendation. The system tries to guess how the user will rate the articles in the database, based on his previous ratings and the ratings of the other users. If the operation is successful then the user receives the top rated articles, otherwise the user is informed of the error.

3.2.10 Get related articles

A GET request is made to the RESTful API in order to get the related articles for an article. The minimum required data is the article id for which we need the related articles. The user can also specify a max number of articles to be returned The system intercepts the request and uses all the similarity methods from the presented workflow to calculate the similarity between every article in the database and the current article. Each article has a value from 0 to 1 meaning the similarity with the article for which we requested the related articles. The top related list is returned is if the operation is successful, otherwise the user is be informed of the error.

3.2.11 Get articles related to a collection

A GET request is made to the RESTful API in order to get the related articles for a collection. The minimum required data is the collection article list. By using the previous presented process of getting an article's similarity score to another article, we get the similarity score of all the articles to all the collection articles. We sum up this scores and return the articles with the top similarity score, if the operation is successful, otherwise the user is informed of the error.

3.2.12 Integration with chrome plugin

If the articles in the current publication do not exist in the database, then the chrome extension adds them by doing specific requests to the application's API. Once all the articles have been added, the user may start a view in which he can see the related articles.

Chapter 4

Database Architecture

In this chapter we present our database, how we represented our entities, why we used this representation and what database operations we require.

The system uses Hbase[2] as it's default database for storing data, but can be easily integrated with any kind of database by passing a java class for working with the database entities.

HBase is an open source, non-relational, distributed database modeled after Google's Bigtable[5] and written in Java. It is developed as part of Apache Software Foundation's Apache Hadoop project and runs on top of HDFS (Hadoop Distributed Filesystem), providing BigTable-like capabilities for Hadoop. It provides a fault-tolerant way of storing large quantities of sparse data (small amounts of information caught within a large collection of empty or unimportant data, such as finding the 50 largest items in a group of 2 billion records).

Hbase is divided into tables. Each table has a name. Then, each table has multiple row keys. Each row key can have multiple families (columns). Each family can have multiple qualifiers. Each qualifier has a value.

Table 4.1: Hbase database format

Row key	Family		Family		Family	
	Qualifier Value	Qualifier Value	Qualifier Value	Qualifier Value	Qualifier Value	Qualifier Value

Due to HBase's table format, we get fast access to our article data. Also, we modeled our entities so that we can take full advantage of the NOsql database.

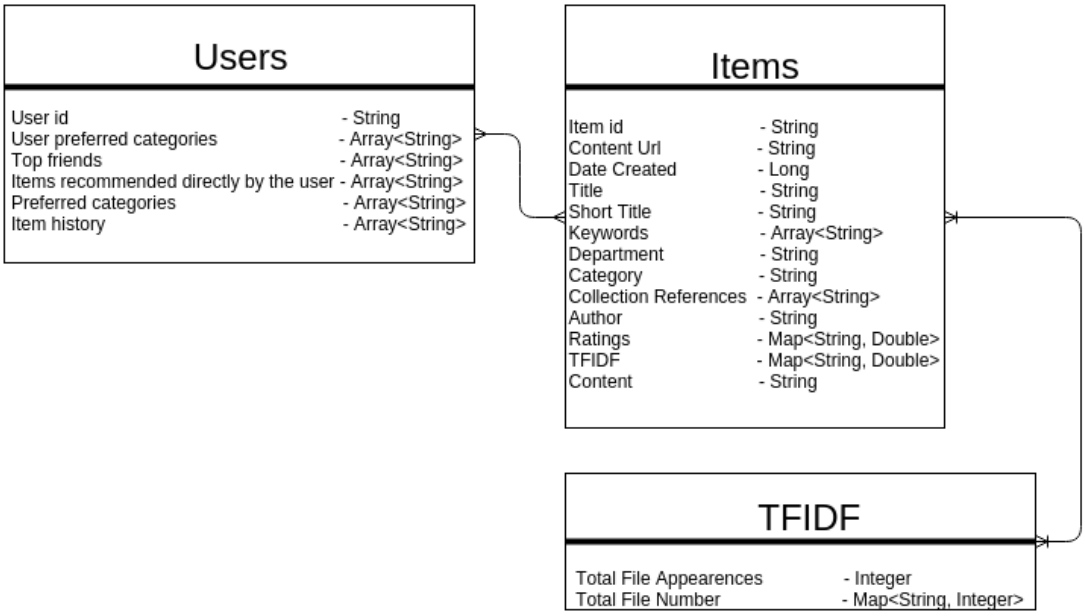
4.1 Database Overview

As we can see in Figure 4.1 we have 3 database tabels: Users, Items and TFIDF. The Users table represents an user in the database and it's required data for being able to make personalised recommendations. The Items table represents an item in our database and it's required data for being able to apply TFIDF to it and make item related recommendations. The TFIDF table holds the required data for applying TFIDF on our items. Each user has a list of items recommended by the user and an item history, referenced by the item's id. Each item has a set of words in it's content and a list of ratings from the users. Each word is saved with the total

[2]http://hbase.apache.org/

number of appearences in all the files, in the TFIDF table. This table also keeps track of the total number of files in the database.

Figure 4.1: Database Schema



4.2 Database entities

We have mapped our database over the attributes that are needed for the recommendation and related algorithms and we present them in the following sections. We will begin with the Users table and end with the Items table, since the TFIDF table isn't a basic required table in order to represent our entities.

4.2.1 Users

These are the basic required attributes in order to represent an user entity.

- user id - String
  We need an user id to uniquely identify each existing user.
- user preferred categories - array of strings
  The user may set certain categories as preferred. The recommendation system will give a greater importance to those categories when making recommendations.

4.2.2 Items

These are the basic required attributes in order to represent an item entity

- item id - String
  We need an unique identifier for each item

- content url - String
  This is an URL or a path which leads to content or information about that item. We need it, in order to be able to access the data of that item and provide recomendations based on similarity.
- date created - date
  This is the date at which the article has been created. We need it in order to make recommendations based on article similarity.
- title - String
  This is the title of an item and it can be even a short desciption. We use this in order to make recommendations based on article similarity.
- short title - String
  This is the short title of an item. We use this in order to make recommendations based on article similarity.
- keywords - array of strings
  This is a set of keywords, set by the item's publisher that represent the item. We use this in order to make recommendations based on article similarity.
- department - String
  This is the department to which an item belongs. It is a more broad term than category. An example of department would be "university" We use this in order to make recommendations based on article similarity.
- category - String
  This is the category to which an item belongs. It is a more precise term than department. An example of category would be "computer science". We use this in order to make recommendations based on article similarity.
- collection references - array of collection URLs
  This is a list of collections to which this item belongs. The publisher may add the item to various collections in order to better organise them and improve the recommendations. We use this in order to make recommendations based on article similarity.
- author - String
  This is the name of the creator of an item. We use this in order to make recommendations based on article similarity.

4.3 Database Design

Besides the basic information, we need to store some extra data in order to speed up the processing time of the recommended and related articles or improve them.

4.3.1 User Table

We need an user table in order to be able to make personalised recommendations for each user.

Table 4.2: User table format

Row key				
User id	Preferred categories	Top friends	Item history	Items recommended directly
	Preferred categories	Top friends	Item history	Items recommended directly
	array list of categories	array list of user ids	array list of item ids	array list of items

- item history - array of item ids
  We need this in order to find the similarity between two users and make collaboative filtering recommendations. The item history may also be used to make recommendations for friends
- items recommended for the user - array of item ids (caching)
  In order to speed up things we cache the items recommended for a user.
- top friends - array of user ids
  The user may add friends to their account. If no friends are selected then a list will be automatically generated by calculating the similarity between different users. We use the friend list in order to make recommendations based on the friend's recommended articles, item history and items recommended directly by that user.
- items recommended directly by the user - array of item ids
  These items have been directly recommended by the user and will be recommended to friends of this user. These items can also be used when doing collaborative filtering, since we know that the user prefers them.

4.3.2 Item Table

We need this table in order to be able to represent an item in the database and not use another system to save them.

Table 4.3: Item table format

Row key						
Item id	Content URL	Date created	Title	Short Title	Keywords	Department
Item id	Content URL	Date created	Title	Short Title	Keywords	Department
String	String	Long	String	String	Array of String	String

Row key						
Item id	Category	Collection references	Author	Ratings		
Item id	Category	Collection references	Author	User id	User id	User id
String	String	Array of collection URLs	String	Double	Double	Double

Row key				
Item id	TFIDF			Content
Item id	Word	Word	Word	Content
String	Double	Double	Double	String

- rating - map of userId, double
  These are the ratings given by users to this item. We use this in order to make recommendations based on collaborative filtering to users. Users that gave simillar ratings to the same item most likely have common interests and preferences.
- TF - map of word, double
  This will store the term frequency value of the top 100 words for an item. We need this in order to speed up the recommandations based on item similarity. This value will be updated after a certain time, since we need the IDF(inverse document frequency) of all the words existing in the database in order to calculate the top 100 words.
- Content - String
  If a content URL can't be provided, then the content can be saved directly, as a string.

4.3.3 TFIDF Table

Table 4.4: TFIDF table

Row key	Row key			
Total file appearences	Item name		Item id	
Total file appearences	Word	Word	Word	Word
Integer	Integer	Integer	Integer	Integer

- Total file appearences - Integer
  We need this in order to keep track of the total number of files existing in the database and to be able to compute the TFIDF
- Item name, Word, Integer
  We need this in order to keep track of the total number of times an word appears in all the files. If the word is used very often then, it will have a lesser importance in the computation of the TFIDF.

4.4 Database Operations

In order to use the database's entities we need a couple of database operations. If we want to use the system with another type of database, besides Hbase, these operations will have to be implemmented for that specific API.

- Check if a table exists
  We need this in order to check if a table exists, before doing any operations on it.
- Create a table
  If the database is new and the table does not exist we need to create it.
- Delete a table
  We need this in order to clear the database and delete all tables, if needed
- Add item to a table
  We need this in order to add entities and data to those entities.
- Delete item from a table
  We need this in order to remove items from the table when they are no longer needed
- Get one item from a table
  We need this in order to access the database's data.

Chapter 5

Testing and Validation

The first step to test and validate our solution was to get an article database. To have that, we filled our database with random values, so that we could check that we configured it correctly and to have a solid base on which we could test our recommendation algorithms. Further on, we used data from different sources, to populate our database and try our algorithms. In the following sections we describe the sources used and the tests that we made.

5.1 Data sources

In order to test all the functionalities of our system we needed multiple data sources, since finding a source with all the required data was not possible. In the following sections we present the data sources used. Some of them are from Adobe's previous project for publishing articles online.

5.1.1 Random Database

This was the first data we added to our database. In order to have data that could be used for NLP we needed to use real words. To do that, we defined a small dictionary, with a couple of hundreds of words. The main purpose of it was to test our database for basic functionality and easily develop the system.

5.1.2 Fast Company

Fast Company[2] is a buiness magazine that focuses on technology, buisness and design. It's online platform uses Adobe's older application and has a database, very similarly in structure to ours. In order to get their data we made calls to their public API which returned JSON objects. Because we needed to get a large amount of data which took a lot of time, we made the API calls and processing in parallel. The API provided us with almost all the data we needed, except the rating of an article and user data. We used it for testing and improving our related algorithms.

[2] http://www.fastcompany.com/

5.1.3 Movie Lens

This[1] data set was provided by GroupLens Research which collected and provided rating data sets from the MovieLens web site. We used a set of data which had 100 000 ratings from 1000 users on 1700 movies. By using that set we were able to predict what rating would a user give to a movie, based on it's previous ratings and the ratings of other users. This data set was used for testing and improving our recommendation algorithms.

5.2 Testing

In order to check that our system works accordingly to our expectations, we ran the following tests, which are described below. We required tests both for our basic operations and our algorithms.

5.2.1 Endpoints

In the following sections we present some testing scenarios that validate the fact that the system's endpoints work accordingly to the use cases.

5.2.1.1 Add User

The Add User operation is a must for our recommendation algorithm. Each user needs to have a profile, based on which we can make article recommendations. To check that the Add User operation works we ran the following tests:

- We tried adding an user without an user id, which resulted in an error message, informing us that this endpoint requires an user id.
- We tried adding an user with an user id, which resulted in a HTTP 200 success message, informing us that the operation was successful.
- We tried adding an user with an user id and multiple data, which resulted in a HTTP 200 success message, informing us that the operation was successful.

5.2.1.2 Add Article

The Add Article operation is a must for our recommendation algorithm. To check that the Add Article operation works we ran the following tests:

- We tried adding an article without an article id, which resulted in an error message, informing us that this endpoint requires an article id.
- We tried adding an article with an article id, which resulted in a HTTP 200 success message, informing us that the operation was successful.
- We tried adding an article with an article id and multiple data, which resulted in a HTTP 200 success message, informing us that the operation was successful.

[1] http://grouplens.org/datasets/movielens/

5.2.1.3 Add User Friend

The Add User Friend operation is required only for getting recommendations based on friends recommendations. To check that the add user friend operation works we ran the following tests:

- We tried adding an user's friend without an user id, which resulted in an error message, informing us that this endpoint requires an user id.
- We tried adding an user's friend without a friend id, which resulted in an error message, informing us that this endpoint requires a friend id.
- We tried adding an user with an user and a friend id, which resulted in a HTTP 200 success message, informing us that the operation was successful.

5.2.1.4 Add User Direct Recommendation

The Add User Direct Recommendation, combined with Add User Friend operation provides us with the means required to make friend's based recommendations. To check that the Add User Direct Recommendation operation works we ran the following tests:

- We tried adding a friend's recommendation without an user id, which resulted in an error message, informing us that this endpoint requires an user id.
- We tried adding a friend's recommendation without an article id, which resulted in an error message, informing us that this endpoint requires an article id.
- We tried adding an user with an user and an article id, which resulted in a HTTP 200 success message, informing us that the operation was successful.

5.2.1.5 Delete User

In order to be able to clear our database, we also need to delete users. To check that the Delete User operation works we ran the following tests:

- We tried deleting an user, without providing an user id, which resulted in an error message, informing us that this endpoint requires an user id.
- We tried deleting an user with an user id, which resulted in a HTTP 200 success message, informing us that the operation was successful.

5.2.1.6 Update User Article History

In order to be able to make recommmendations based on collaborative filerting, we needed an user article history, which records all the articles read by the user. To check that the Update User Article History operation works we ran the following tests:

- We tried updating the article history of an user, without providing an user id, which resulted in an error message, informing us that this endpoint requires an user id.
- We tried updating the article history of an user, without providing an article id, which resulted in an error message, informing us that this endpoint requires an article id.
- We tried updating the article history of an user with an article id, which resulted in a HTTP 200 success message, informing us that the operation was successful.

5.2.1.7 Get Friend's Article Recommendations

This endpoint allows us to get an user's article recommendations based on his friends. To check that the Get Friend's Article Recommendations operation works we ran the following tests:

- We tried getting the friend's article recommendation of an user, without providing an user id, which resulted in an error message, informing us that this endpoint requires an user id.
- We tried getting the friend's article recommendation of an user with an article id, which resulted in a HTTP 200 success message, informing us that the operation was successful.
- We checked that we are getting the correct answer by adding a friend and a friend's recommended article and that we are getting the added article.

5.2.1.8 Get Articles

This endpoint allows us to get a certain number of articles from the database. To check that the Get Friend's Article Recommendations operation works we ran the following tests:

- We added a couple of articles, by using the Add Article operation, after which we checked if the Get Articles operation returns the added articles.

5.2.1.9 Get Recommended Articles

This endpoint allows us to get the recommended articles for an user, based on his history. To check that the Get Recommended Articles operation works we ran the following tests:

- We tried getting the recommended articles, without providing an user id, which resulted in an error message, informing us that this endpoint requires an user id.
- We tried getting the recommended articles of an user with an user id, which resulted in a list of articles recommended for that user in JSON format.

5.2.1.10 Get Related Articles

This endpoint allows us to get the articles related to another article, based on it's attributes. To check that the Get Related Articles operation works we ran the following tests:

- We tried getting the related articles, without providing an article id, which resulted in an error message, informing us that this endpoint requires an article id.
- We tried getting the recommended articles with an article id, which resulted in a list of articles related to that article, in JSON format.

5.2.1.11 Get Articles Related to a Collection

This endpoint allows us to get the articles related to a collection, based on it's attributes and the attributes of the articles in the collection. To check that the Get Articles Related to a Collection operation works we ran the following tests:

- We tried getting the articles related to a collection, without providing any of the collection's attributes, which resulted in an error message, informing us that this endpoint requires the collection's attributes, which are simillar to that of an article.

- We tried getting the related articles with a collection's attributes, which resulted in a list of articles related to that collection, in JSON format.
- We tried getting the related articles with a collection's attributes and a list of articles ids which are in that collection, which resulted in a list of articles related to that collection, in JSON format.

5.3 Evaluation

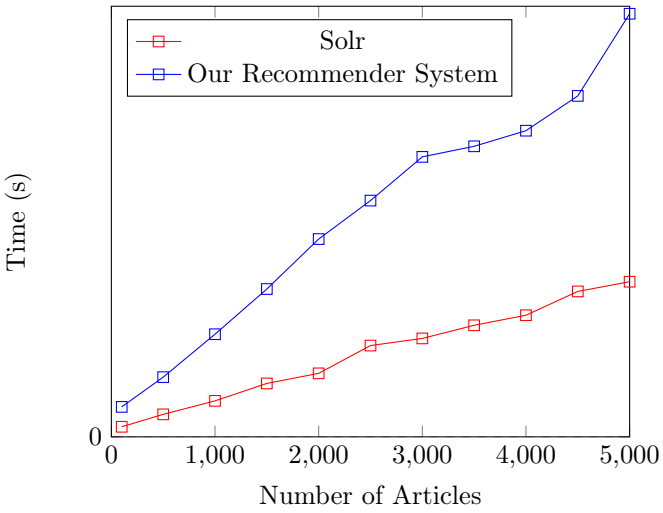
In this section we describe how we evaluated our system and how well it performs. We can apply three types of experiments, in order to evaluate our recommender's quality [3]. The first one is an offline setting, where recommendations are compared without user interaction. The second one is based on user studies, where a group of subjects interact with the system and report on the experience. The third one and the best is represented by large a scale, online experiment, where we test the system in a production environment.

5.3.1 Time and Memory Usage

In the following subsections we make an analysis of the time and memory usage of our system, when applying the related and recommended algorithms, compared to solr.

5.3.1.1 Article Indexing Time

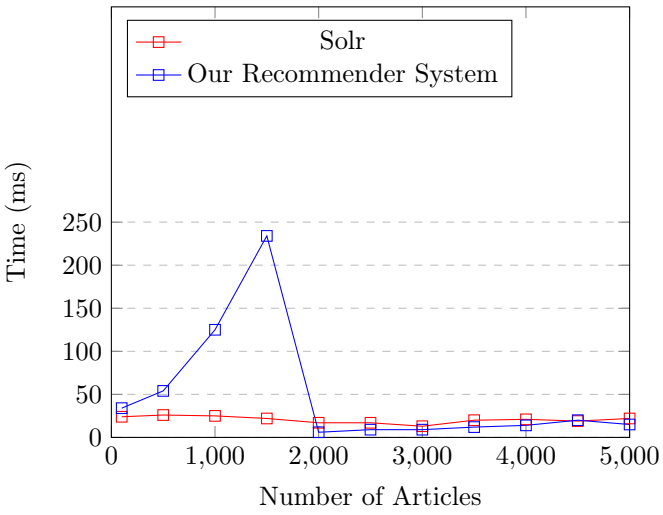
Figure 5.1: Article Indexing Time



Indexing time refers to the amount of time needed by a system to add a new item or a group of items to it's data. We have to index items in order to make the system aware of the existing information and to make certain optimisations that lower our response time. In the above image, Figure 5.1, we can see our and solr's performance on item indexing. Our time is slower than that of solr because we use a database to store our data. This allows us to keep it in a structured manner and provide personalised recommendations, a feature that solr does not have.

5.3.1.2 Related Article Time

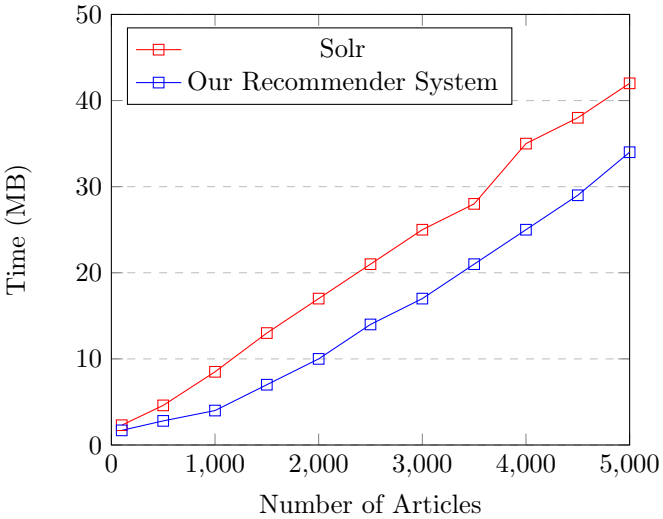
Figure 5.2: Related Article Time



In Figure 5.2, we can see how solr's and our related article recommendations perform. In the begining of the experiment we didn't use any form of caching, thus using less RAM memory but we got a response time that is worse than that of solr. Then, we started using our caching mechanisms and we got a simillar time.

5.3.1.3 Disk Memory Usage

Figure 5.3: Disk Memory Usage



In the above image, Figure 5.3, we can see how solr's disk usage compares to ours. We can clearly see that we use less memory than solr. Even though, this may cause some speed issues, when using it without caching, our system is ideal for applications that have a limited amount of memory at their disposal.

5.3.2 Quality of Results

In the following subsections we make an analysis of the quality of the results given by our system, when applying the related or recommended algorithms.

5.3.2.1 Related Articles

Because it was almost impossible to check if we got good and relevant results with a random database we had to take a database from a previous project of Adobe, which was used by one of their clients, Fast Company[1].

After we populated our database with their data, we started running tests on it by changing the importance of each attribute and printing the top 100 results

In order to check that we were obtaining good results we chose an article and google searched it's title on fast company's site. We then classified the outputs in 3 categories, by relevance, and saved the data in an expected file.

Using the expected data we then binary searched for the best importance of each attribute. Because the expected data that we chose may have been influenced by subjective factors, we decided to test our recommendation system by using solr.

Using the best importance values we determined by using the expected file, we got our ten most related articles in solr's top 25 related.

5.3.2.2 Recommended Articles

In order to test that the system gives good recommendations we took the dataset of MovieLens which provided 100,000 ratings from 1000 users on 1700 movies. Using that data we then used our algorithms to predict what rating would a user give to a certain movie, already rated, and compared it to the actual given rating. By doing this we obtained a deviation from the removed rating of about 15%.

[1] http://www.fastcompany.com/

Chapter 6

Conclusions and Future Development

In this chapter we present the end conclusions of this thesis and the possible future development of it.

6.1 Conclusions

In this thesis we built a stand-alone application that can give good recommendations and can be easily integrated with any database. A system that takes full advantage of all the existing data and gives the best recommendations for the presented problem.

Also, the importance of each considered attribute can be easily changed and thus we can easily make recommendations of articles made by a certain author, belonging to a certain category, belonging to the same collection, etc. We can combine the recommended articles for a certain user with the related articles, in order to solve the cold start problem and to obtain better results.

6.2 Future Development

In order to further improve the recommendations we could use a neural network to predict what ratings a user would give to the unrated articles[6]. This way, we could better predict the users with similar interests. Artificial neural networks are a family of statistical learning models inspired by the biological neural networks, in which a group of neurons work together to predict the outcome of an action, based on their previous experience. The connection between each of the neurons has a certain weight that can be changed and thus the network can adapt and learn based on it's previous experience. An artificial neural network has to be trained before being used. This means that we need to give it both an input data set, which will go through the network and produce results, and an output data set, which is the expected result. The neural network will adjust the weights between the neurons in order to get closer to the output data set. After the training stage, the neural network will be capable of predicting the result, based on it's training set.

In our case, we can train the neural network to predict what rating a user will give to an item, based on it's other ratings and the similarity of the rated items. By having that rating, we could improve the collaborative filtering results, since we can predict the ratings a user would

give to all the items. By having that prediction we can better find users with simillar interests and make recommendations based on that.

Another great feature would be to set the related articles algorithm to adjust the importance it gives to each attribute based on an user's input. This means that if, for example users tend to choose and spend more time on articles, from the related articles list that have a greater similarity on the category attribute, then that attribute's importance will increase by a certain amount and the other attributes' importance will adjust as well.

Bibliography

[1] Drakoulis Martakos Charalampos Vassiliou, Dimitris Stamoulis and Sortiris Athanassopoulo. A recommender system framework combining neural networks and collaborative filtering. http://www.wseas.us/e-library/conferences/2006hangzhou/papers/531-656.pdf, April 2006.

[2] Brent Smith Greg Linden and Jeremy York. Amazon.com recommendations item-to-item collaborative filtering. http://www.cs.umd.edu/~samir/498/Amazon-Recommendations.pdf, February 2003.

[3] Shani Guy and Gunawardana Asela. Evaluating recommendation systems. http://research.microsoft.com/pubs/115396/EvaluationMetrics.TR.pdf.

[4] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. http://nlp.stanford.edu/pubs/StanfordCoreNlp2014.pdf, April 2014.

[5] Chang MFay, Dean Jeffrey, Ghemawat Sanjay, Hsieh Wilson, C., Wallach Deborah, A., Burrows Mike, Chandra Tushar, Fikes Andrew, and Gruber Robert, E. Bigtable: A distributed storage system for structured data. http://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf, 2006.

[6] Yehuda Koren Robert M. Bell and Chris Volinsky. The bellkor 2008 solution to the netflix prize. http://www2.research.att.com/~volinsky/netflix/Bellkor2008.pdf, June 2008.

[7] Michael R. Smith and Tony Martinez. A hybrid latent variable neural network model for item recommendation. http://arxiv.org/pdf/1406.2235.pdf, June 2014.