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
Sl.Dr.Ing. Razvan 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 Razvan Deaconescu
for his unending support in the redactation
of the bachelor thesis.

Abstract

This thesis presents a recommendation system for articles that are rich in content and have certain attributes. This system was needed because a tool that exploited all the properties of an article, at their maximum potential, was not that accessible. 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 article categorization. Other recommendation systems take into account only the content of the articles and do not give any importance to the date, author, language, and ratings of an article. Also, most of the already existing recommendation systems create and maintain the data in their own database, thus doubling the required storage space for an application.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recommendation systems have become a major research area since the appearance of the first paper on collaborative filtering in the mid 1990s and have become popular in both commercially 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 take into account only the text of the articles and do not give any importance to the date, author, language, and ratings of an article. Also, most of the already existing recommendation systems create and maintain the data in their own database, thus doubling the required storage space for an application. In this thesis we will describe a recommender system that solves all the mentioned problems of the existing systems.

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 precise recommendations.

   This type of recommendations should make use of all the attributes of an article and give precise 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 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, I will present a couple of the most successful recommendation systems at the moment.

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

These are the most famous type 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 all the indexed websites using a certain phrase or combination of words.

The one that offers a tool most similar to an articles recommendation system is represented by advanced Google search. 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 is an open source enterprise search platform, written in Java, from the Apache Lucene project. Its 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. 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 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	✓	✓	✓	✓	✓

TODO DELETE THIS: E VREO PROBLEMA CA M-AM CONCERNTRAT PE CE REZOLV EU SI NU REZOLVA EI SI NU INVERS?

Chapter 2

Recommendation Algorithms

There are 3 types of recommendation algorithms:

1. Recommendations based on content

   These 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 a similarity between two items.

2. Recommendations based on collaborative filtering

   These 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 the users.

3. Hybrid recommendations

   These 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 are going to present the implemmented algorithms.

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 a kind of article grouping by user preferences. Usually, users fit into a certain profile and will read articles that may be 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 will 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 derivational affixes.

   For instance:

   (a) The word "*am*" will be reduced through lemmatization to the base form "*be*". This form can not be deducted 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 deducted 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 may not be correct in certain contexts.

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

   TODO Write more about it

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 [4]

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

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

Database Architecture

The system uses Hbase 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 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. That is, 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 3.1: Hbase database format

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

We are working with hbase because it grants us fast access to our article data.

3.1 Database entities

We have mapped our database over the attributes that are needed for the recommendation and related algorithms.

3.1.1 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.
- 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.

3.1.2 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 precise term than category. An example of department would be "computer science". 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 broad term than department. An example of category would be "university" 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.

3.2 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.

3.2.1 User Table

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

Table 3.2: 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.

3.2.2 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 3.3: 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

- 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.

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

- 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.

3.2.3 TFIDF Table

Table 3.4: 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.

3.3 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 4

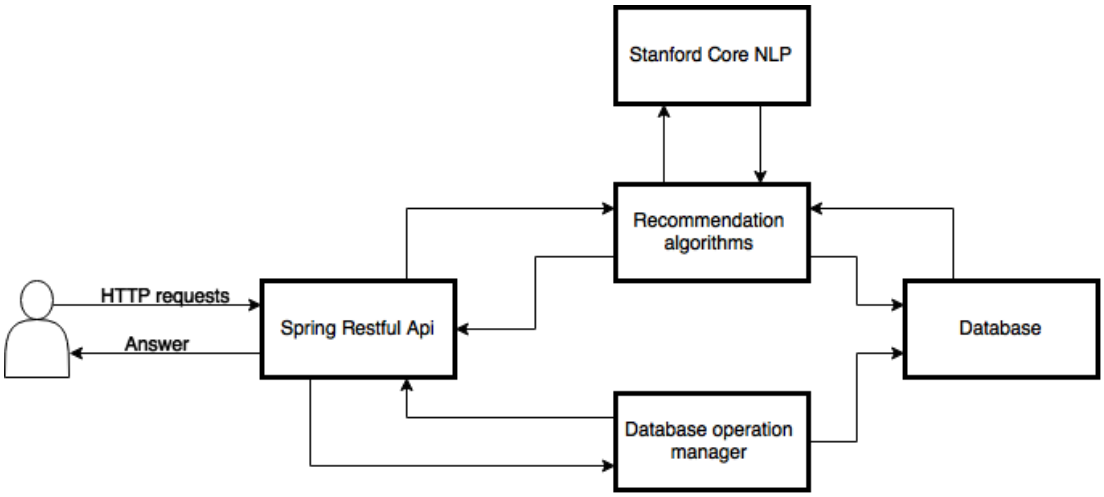
System Architecture and Workflow

In the following sections we are going to talk about the various architecture and technology choices.

Also, we are going to present the workflows of the system.

4.1 Recommendation System Architecture

Figure 4.1: System Architecture



4.1.1 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.

4.1.1.1 Java

Java 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 used 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.

4.1.1.2 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 commonly usage of Javascript is as a client side scripting language. This means that the Javascript code is send 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 used 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 Node.js was used in order to create a proxy between our Recommender System and the Javascript client. Also, Node.js was used to serve the files to the client.
Also, in order to better show our Recommender System's capabilities, a Chrome plugin was built 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.

4.1.2 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.

4.1.2.1 Spring Framework

4.1.2.2 Apache HBase

Hbase is an open source, non relational and distributed databse, modeled after Google's Big Table. 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:

- 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 chosing HBase was the fact that it has a great community and support.

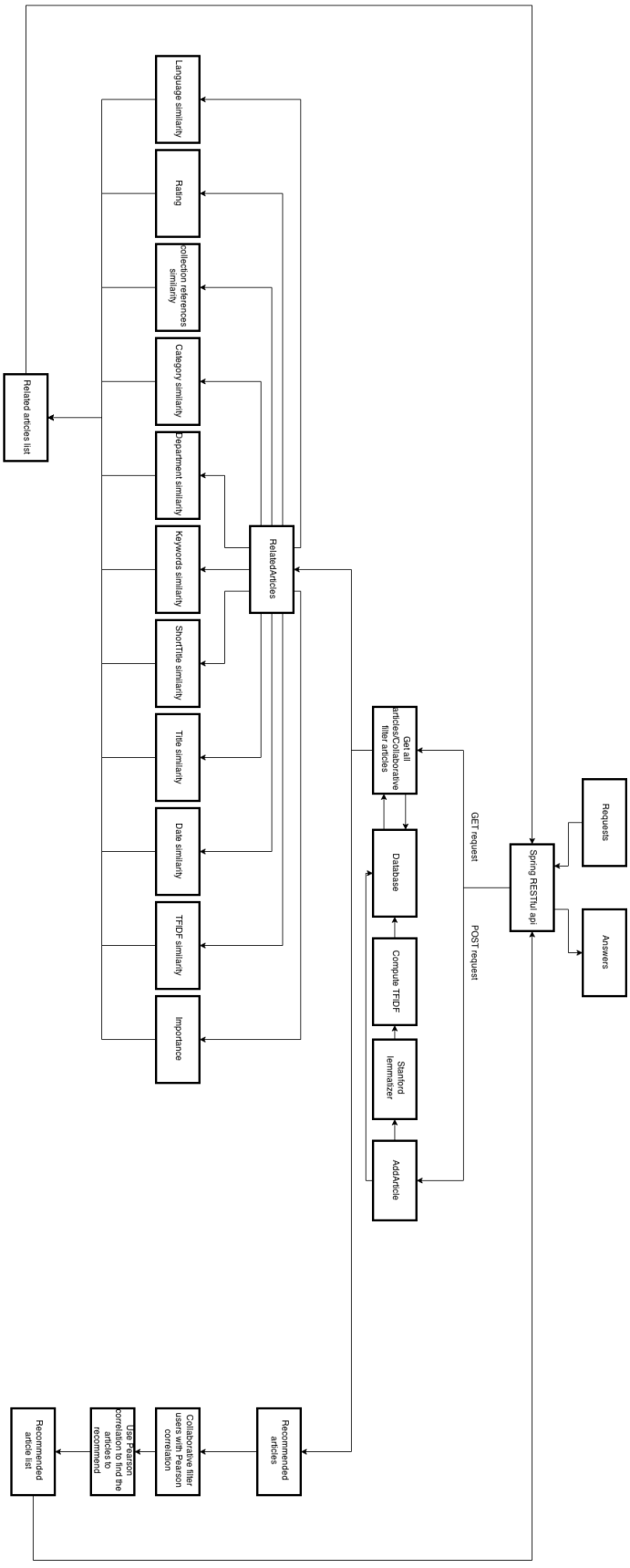
4.1.2.3 Stanford CoreNLP

- Present Spring and why we chose it
- Present Java and why use chose it
- Present HBASE and why we chose it(not sure about this. Small presentation in database design. Can bring it from there)
- Present Stanford lemmatizer and why we chose it.

4.2 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 are going to talk about the implemented use cases.
All the data will be returned as a JSON object.

Figure 4.2: System Workflow



4.2.1 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 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 will receive "true", otherwise the user will be informed of the error.

4.2.2 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 will receive "true", otherwise the user will be informed of the error.

4.2.3 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 will receive "true",otherwise the user will be informed of the error.

4.2.4 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 will receive "true", otherwise the user will be informed of the error.

4.2.5 Update user article history

A PUT request is made to the RESTful api with the required data for updating an user 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 will return "true", otherwise the user will be informed of the error.

4.2.6 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 will receive "true", otherwise the user will be informed of the error.

4.2.7 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 will receive "true", otherwise the user will be informed of the error.

4.2.8 Get all articles

A GET request is made to the RESTful api in order to get all the articles.
The user may specify how many articles he wishes to receive.
The articles are fetched from the database and returned to the user.
If the operation is successful then the user will receive the article list, otherwise the user will be informed of the error

4.2.9 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 will try to guess how the user will rate the articles in the database, based on his previous ratings and the ratings of the other users he has the most common interests.
If the operation is successful then the user will receive the top rated articles, otherwise the user will be informed of the error.

4.2.10 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 may also specify a max number of articles to be returned
The system intercepts the request and will use all the similarity methods from the presented workflow to calculate the similarity between every article in the database and the current article. Each article will have 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 will be informed of the error.

4.2.11 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 was successful, otherwise the user will be informed of the error.

4.2.12 Integration with chrome plugin

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

- Add the workflow
- Add all the workflows(use cases) and explain them in depth

Chapter 5

Testing and Validation

In order to test and validate my solution I needed an article database. To have that, I filled my database with random values, so that I could check that I have configured it correctly and to have a solid base on which I could test my recommendation algorithms.

5.1 Testing

TODO Add al the used databses (fast company, national geographic, etc)

5.1.1 Basic Operations

TODO

- Add the test cases for the basic operations, unrelated to the next two sections

5.1.2 Related Articles Operations

TODO

- Add the test cases for related articles.

5.1.3 Recommended Articles Operations

TODO

- Add the test cases for recommended articles.

5.2 Evaluation

TODO

- Write why it is hard to evaluate the recommender system.
- Write the evaluation scenarios that do not depend on related or recommended.

5.2.1 Related Articles

5.2.1.1 Final Result

Because it was almost impossible to check if I got good and relevant results with a random database I had to take a database from a previous project of Adobe, which was used by one of their clients, fast company.

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

In order to check that I was giving a good result I chose an article and google searched its title on fast company's site. I then classified the outputs in 3 categories, by relevance and saved the data in an expected file.

Using the expected data I then binary searched for the best importance of each attribute. Because the expected data that I chose may have been determined by my own personality, I decided to test my recommendation system by using solr.

Using the best importance values I determined by using the expected file, I got my 10 most related articles in solr top 25 related.

In order to further confirm that my resulted articles are really related I built a simple web page in which users chose one of 4 degrees of relatedness and the data was saved on a server.

5.2.2 Recommended Articles

5.2.2.1 Final Result

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%.

Chapter 6

Conclusions and Future Development

6.1 Conclusions

We proved in this thesis that we built a stand-alone application that can give good recommendations and can be easily integrated with any database.

We built 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, belong to a certain category, belong 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[3]. This way, we could better predict the users with similar interests. TODO

- Add more about the neural network.

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] 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.

[4] 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.