Insider

Documentation

TDT4215 – Web Intelligence

Albert Hambardzumyan
Trong Huu Nguyen
Audun Arnessønn Sæther

Group 17

Spring 2016

Table of Contents

Introduction 3

System Architecture 4

- RESTful API 4
- Sentiment Analysis 5
- Collaborative Filtering 6
  - Discussion of Potential Issues 6
- Data Source 7

How It Works 8

- Home Page Overview 8
- Item Page Overview 8

How to Configure and Run 10

Challenges 13

Conclusion and Remarks 14

References 15

Introduction

Our project idea was to make a form of a travel destination recommender system, where a user could input a destination, landmark, hotel, etc. and be presented with basic information as well as see reviews left by other users and a score based on these reviews by using sentiment analysis. We also wanted to provide two types of recommendations for the user, the first being nearby spots sorted by their rating in descending order. The other type of recommender system we implemented was a collaborative filtering based on the users own reviews and ratings as well as the ones left by other users. Due to time constraints and lack of data sources our system will only deal with hotels for one city, but it should be able to handle other types of items for future work.

We might have taken on a task that was more than required, especially with combining both sentiment analysis with two types of recommender systems. However, it did work out in the end and we are happy with the results.

System Architecture

The system consists of 4 parts: website and web service, Python module, and Java module. The web service is representational state transfer service (called RESTful). The web service does the business logic part and gives an API (Application Program Interface) to clients for making request. For the RESTful API we used the most popular technology for that at this moment, namely NodeJS. For the website itself we have used AngularJS.

The Python module does the semantic analysis and stores the results in the database. The Java module deals with collaborative filtering, and again stores the results in the database. Below are the detail description of the Python and Java modules.

RESTful API

Below is the API interface of our project:

Interface	Path	Description
/search	/	Names of the items for searching. Used for autocomplete search purposes.
/hotels	/	Names of all hotels.
	/:hotelId	All information of the hotel.
	/:hotelId/rate/:rate/users/:user	Post the rate of the particular item for the particular user.
	/:hotelId/users/:user	Get the rate of the particular item for the particular user.
/recommendCf	/ :userId	Recommended items for a particular user computed at offline stage.

/recommendClosest	/ :hotelId	Recommended items closer to the particular hotel and which has highest rank.
/sentiment	/ :hotelId	Does sentimental analysis for a particular item. Based on the analysis a rank is calculated and stored in the database. Route is designed for administrative purposes.
/collaborativeFiltering	/ :userId	Does collaborative filtering for a particular user. All the results is stored in the database. Route is designed for administrative purposes.
/python	/	Does sentimental analysis on an empty string. Usage: quickly figure out whether the python shell works or not.

Sentiment Analysis

The sentiment analysis module relies heavily on TextBlob[1], which is a Python library that provides an API for various text analysis tools, including noun phrase extraction, part-of-speech tagging, sentiment analysis among others.

For our project, we were interested in primarily using the sentiment analyzer, which we did with a basic Python script that parses an input string, converts it to a TextBlob object, and by using the standard implementation of TextBlob's sentiment analyzer it returns the sentiment polarity of the given string (Positive, Neutral or Negative).

The default sentiment analyzer in TextBlob is based on the sentiment analyzer defined in the pattern.en library [2], and although not perfectly accurate will suffice for the scope of this project. This module works by using a set of defined lexicons and associated polarity scores of adjectives that frequently occur in reviews, which it will attempt to match to words and phrases in a given string, and in the end average these for the entire string for a total sentiment value [3], [4].

[1] https://textblob.readthedocs.org/en/dev/

Collaborative Filtering

The collaborative filtering is done using the Apache Mahout library[2] . We chose this recommendation engine because one of the group members had previous experience using it. In addition, and in contrast with a lot of other recommendation engines, it is an active project with constant updates and improvements. Mahout offers methods both for user-based and item-based collaborative filtering. We chose to go with the user-based approach, as the main argument for choosing the item-based approach over user-based is scaling (Jannach et al. 2011). Scaling is not that important in this application, considering the low number of hotels and users. The users are created by us, and amount to 100 users. Each of these user has rated 15 randomly chosen hotels. Ideally, we should have had a set of ratings on the hotels from real users, but we could not find this.

The recommendation process consist of a number of steps. First, the all the ratings on hotels is loaded into a data model. Based on this data model similarities between users is calculated, using Mahouts PearsonCorrelationSimilarity-method. Then, a neighborhood for the user that should get recommendations is created based on the similarities found in the previous step. A user is included in the neighborhood if the similarity value computed is above a given threshold. We have set this threshold to 0.1. This may seem like a low similarity value, but given the low number of users and randomness in ratings it is necessary to ensure that enough users is included in the neighborhood so that recommendations can be made. The recommender fetches all the hotels that the users in the neighborhood has rated, except for the hotels the user receiving recommendations has already rated. Finally, an estimation of how the user will rate the unrated hotels in the neighborhood is made, and the top-n recommended hotels is returned and stored in the database. The n is set by us, and is set to 7 (the reason for this number is related to the presentation front end). [3]

Discussion of Potential Issues

A normal problem with collaborative filtering is the cold start problem, which is the problem of recommending items to a user that has not rated any items. In our application, all users in the system starts with 15 ratings and it is not possible to create new users, and as such the cold start problem is not an issue.

As we have chosen to randomly choose which hotels to rate for each user and what the rating should be, the rating patterns may not resemble how real humans would rate the hotels. Collaborative filtering is about finding patterns in ratings and suggest new items based on the patterns. That is, if users have similar tastes in the past the assumption is they also will have the similar tastes in the future (Jannach et al. 2011). Given that the past

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

[3] The Mahout code used can be found at:
https://github.com/apache/mahout/tree/master/mr/src/main/java/org/apache/mahout/cf/taste

for a user is set randomly, it may be tempting to ask if the recommendations become random. However, the past is changeable, and as the user rates more hotels the random ratings will have less and less importance on the recommendations (the user can also change the already set ratings). Therefore, the recommendations should get better over time, at least until it reaches a tipping point where the most interesting hotels is already rated by the user and is thus not possible to recommend any more.

Because of the relatively small number of users and ratings, the neighborhood of a user may be small. Consequently, the number of unrated hotels for a user in a neighborhood may be small. This may result in recommendations that are not particularly good (i.e. low predicted rating) and/or fewer than expected recommendations, because such recommendations are the only available. The decision of 15 start ratings for each user is a compromise between avoiding the cold start problem, while at the same time giving users a chance to influence the future recommendations (with too many start ratings, the "real" ratings would have less influence on the recommendations).

Generally, the recommender could need some testing and fine-tuning. For instance, the threshold value could have been experimented with, to see what value of this gives both good and enough recommendations. Mahout also provide some testing methods that splits the data set in a training set and a test set, and evaluates how well the recommender can predict the ratings in the test set based on learning from the training set. These methods could have been used to for instance test different set of random rankings and find the best one.

Data Source

The data we found and used in our system is a subset of the OpinRank Dataset by K.A. Ganesan and C. X. Zhai (available in TSV format at [2]). The original data set contains reviews for cars and hotels in ten different cities, scraped from Edmunds and TripAdvisor, respectively. Additionally, the data includes basic information of each hotel, among them name, street, user-scores (though this is not used as we wanted to test our own sentiment analysis module). The reviews themselves are anonymized, and only include date, title and full reviews.

As we were only interested in the hotel part, as well as having to manually sanitize and modify the data set for our usage, we focused on hotels in one city: New York. Our final subset of the data set comprises 259 hotels, with 33861 feedbacks/reviews related to these hotels, which is 130 feedbacks for each hotel on average.

How It Works

Home Page Overview

As the application loads it requests for a user id to be inputted. Note that this can be skipped, but in that case there will not be any collaborative filtering recommendations.

Next, a search box will appear and allows searching for types of items. The items will be loaded from the database after the desired items is specified. Note, within this project scope we are dealing with only one type of the item, which is “Hotel”. This is to limit the scope of this project, and because we found a good source of data for hotels in New York. This is described in the “Data Source” section. For future reference, the system is able to handle other types of items, e.g. tourist attractions, cities in general etc.

Clicking on one of the loaded items redirects to another page where the detailed information of that item is shown.

Item Page Overview

As the page loads, it takes the ‘hotel id’ and ‘user id’ stored in the local storage by previous page. Based on this data, four independent requests goes to the server side to resolve the data related to this item and user.

The ‘Rank’ value indicates how popular the selected item is, and the range of its value is between 1 and 5. The calculation is done in offline stage using the feedback data. The sentiment analysis module returns the number of positive, negative and neutral values. Based on the sentiment analyses, a simple arithmetic is done to calculate the ‘Rank’ value. The formula is: positive / total * 5. More details description of the Sentiment Analysis module can be found in the ‘Sentiment Analysis’ section above. The sentiment analysis part can be tested directly using the admin panel (see the ‘How to configure and run’ section).

The stars in the ‘Give a rate’ component indicates the user’s previous rating for this item. If the item has not been rated by the user, no star is filled and the user can rate the item. As a result of rating a hotel, a new call will be made to the server and the rating score will be stored in the database, which will be taken into account the next time collaborative filtering is done.

The map section takes the coordinates of the item, and shows the item on the map. It sounds simple, however our original data source lacked the coordinates for the hotels, and we had to manually go through and add these. It was necessary not only for the map section, but also for computing recommendations for closest hotels with highest rank.

The ‘Nearby ones with highest rank’ works in the following way: a message is passed to the server; the server then takes all the hotels and computes the distance between the item and other ones. All items within the range go to next stage of filtering. Allowed distance number is specified in a config file, and can be easily changed. Currently it is set to 1 km. In the next stage, hotels are filtering by their rank. First, we take all of the hotels having rank between 4 and 5. If the number of such hotels does not satisfy the minimum allowed number, then we consider ones with rank 3 and 4 too. This continues until we get the specified amount of recommendations, which is 7. This number of recommendations is specified in a config file and can be changed easily as well.

The collaborative filtering section shows recommendations based on the user and item data. The collaborative filtering module returns items that the particular user can be interested in. More details description of the Collaborative Filtering module you can find in the ‘Collaborative Filtering’ section. Like with the sentiment analysis, the admin panel can be used to test it from the client side.

Clicking on any of the recommended items will redirect to the page of that item. All the undergoing processes are available for inspection in the console both in the client and server sides.

Due to our focus in finding data more appropriate for sentimental analysis and recommendations, our data does not include images for each item. Thus, the image section is static.

How to Configure and Run

Requirements

Install python, make sure classpath is set, and that it runs
`$ pip install -U textblob`
`$ python -m textblob.download_corpora`
Install Java, make sure classpath is set, and that it runs
Install NodeJS
Install MySql

Running

Make configuration changes in `api/app/config/config.js`
Important changes - path: the directory of the project, db configs
Import the `api/app/db/db_dump.sql` file to your database
Download dependencies - `npm install`
Run - `env=dev npm start`

Testing

Main route - http://localhost:3000/api/v1/

Python script test
http://localhost:3000/api/v1/python
Result - result :{"positive":0,"negative":0,"neutral":1}

Testing using admin panel
Username: admin
Password: admin

Sentiment Analysis
Open the item page to check the rank value
Before doing sentiment analysis, set the rank of the item to be zero:

```
update hotels set rate=0 where
id="usa_new%20york%20city_3_west_club";
```

Check it by refreshing the item page.
Do sentiment analysis.
Refresh the item page. The item should have a rank.

Collaborative Filtering
Open the item page to check that there are some recommendations for this user
Before doing collaborative filtering, delete recommendations for that user:

```
select * from recommendations where userId=1;
```

You see some recommendations. Now, delete them:

```
delete from recommendations where userId=1;
```

Check that there is no collaborative recommendations by refreshing the item page. Then, do collaborative filtering.
Refresh the item page. The user should have recommendations

based on collaborative filtering.

Note, all the undergoing processes can be seen in the console, both on the client and server side.

Challenges

One of the first challenges was the finding a decent source of data. There were difficulties in finding proper sources of data that were applicable for our system. The data we eventually found was adequate, though incomplete for our purposes, and we had to manually find and enter the coordinates for all of hotels. Additionally, there were a lot of unnecessary information and reviews unrelated to our hotels, and we had to spend a lot of time on sanitizing the data for use in our database.

Although the system consists of mainly two pages, it is quite complicated and a lot of data has to be transferred. For this reason, there was a need to create client-server communication. Both server and client side code is done in a way that it can be easily extended in the future. We spent a lot of time to set up appropriate structure on both client and server sides, and both of them are satisfying the standards of real product applications. Each piece of logic is separated in one module. Thus, the application consists of a number of services, controllers, templates, models, and configuration files.

The sentiment analysis for 259 hotels with 33861 feedbacks related to these hotels requires high computation cost. It was not possible to analyze all of them at once, due to machine CPU and RAM limitations. As a result, we had to do it for a maximum of 4-5 hotels at a time, which requires hard manual labor.

We ran into the same problem with collaborative filtering, and we had to do it manually for sets of 100 registered persons at a time, as our computer power is limited to do it for all the users at once.

We wanted the application to be interactive, and consequently it results in added complexity. On the other hand, we have ended up with a system that has many features, e.g. a rating feature, and an admin panel to demonstrate from the client side.

Another challenge was using the different technologies (Angular, Node, Python, and Java as mentioned earlier) and integrating them in such a way that they function as a coherent system.

Day by day, the code got longer and more complex, and in the end, we had thousands of lines of code. It was difficult to refactor and document it. However, we ended up with a pretty nice and scalable system. Also, we tried to make it as friendly as possible for demonstration purposes, providing logs of the each main actions.

Conclusion and Remarks

In the end, our system does what we intended it to do. Therefore, we are happy with the result. While we took on a lot of work on the project, maybe too much, we learned a lot as well. Doing both sentiment analysis and two types of recommender systems gave us valuable practical experience and insight in two of main themes in this course.

For future work, the system should be tested more thorough. In particular, the collaborative filtering part needs testing with real users to be improved. Regarding the web page itself, more info on each hotel could be fetched and presented to give a more complete presentation. Such additional info should also improve the collaborative recommender, as the user easier could distinguish between different hotels and thus make more informed ratings.

References

[1] Jannach, Dietmar et al. *Recommender systems: an introduction*. Cambridge University Press, 2011.

[2] TextBlob Documentation. *Sentiment Analyzers*. Available online: http://textblob.readthedocs.org/en/dev/advanced_usage.html#sentiment-analyzers

[3] Schumacher, A. *TextBlob Sentiment: Calculating Polarity and Subjectivity*. 2015. Available online: http://planspace.org/20150607-textblob_sentiment/

[4] pattern.en Documentation. *Sentiment.* Available online: http://www.clips.ua.ac.be/pages/pattern-en#sentiment

[5] Ganesan, K.A., Zhai C. X. *Opinion-Based Entity Ranking*. Springer, 2012. Available online: http://kavita-ganesan.com/entity-ranking-data