UPC - FIB

A Recommender System

Open Data Project Delivery

Víctor Herrero, Diego Garcia-Olano, Elsa Mullor

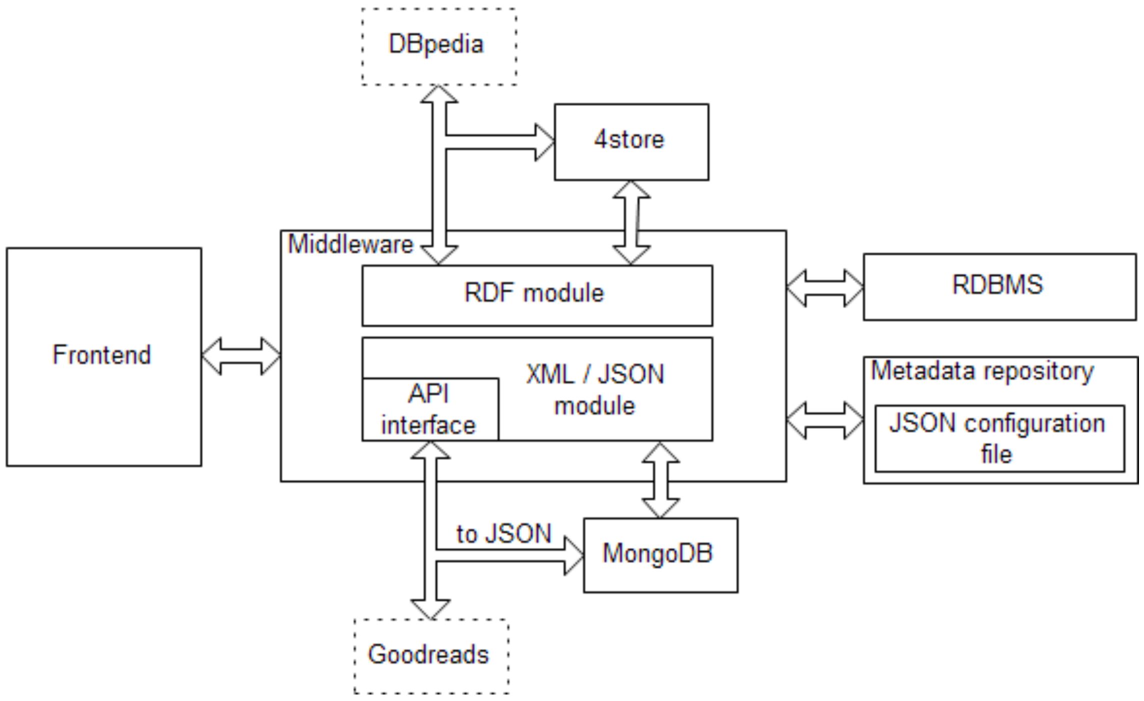
Table of Contents

Data Sources & Sketch of the System Architecture

Data Sources to be integrated into System

Tentative sketch of the system architecture

The following picture depicts a simple architectural diagram of the main modules that will take place in the system.



The frontend will consist of a website and will essentially have two main states: ***not logged in*** vs ***logged in***. The ***not logged in*** state will only be able to run cold start recommendations, since the user will not be identified. The ***logged in*** state should be able to deal with cold start and not cold start recommendations: the former when the user is logged in but we do not know about his likes and the latter when we do.

The middleware will consist of distinct modules which will each be responsible for dealing with one different data source. In addition, the middleware will be based on Python as programming language since it is full of tools and drivers to communicate with the databases. The recommendation service (and others if required) will be exposed as REST services so the middleware is not too coupled with the frontend.

The middleware system will additionally contain two databases:

- 4store as triplestore containing RDF data.
- MongoDB as document-oriented store containing JSON data.

Thus in the middleware, the fields of the both data sources will be mapped to values our frontend will accept for search/display logic.

The middleware will also contain a simple config JSON to know what fields are expected to support and a RDBMS. The goal of the JSON file is that, if we want to add information to the frontend view, we only need to add the desired information to that file. The front end will not then need to know about the internals of the middleware system. The task for the RDBMS will be to provide fast lookups to find what metadata is needed to answer a query, so the JSON config file accesses will be then more optimal. The middleware additionally will be in charge of the API calls to Goodreads and DBpedia, if no local version exists.

Initial data sets

Information for books available in data from:

http://challenges.2014.eswc-conferences.org/index.php/RecSys#DATASET

There are three datasets in flat tab separated values files:

1. *DBbook_Items_DBpedia_mapping.tsv*
2. *DBbook_train_binary.tsv*
3. *DBbook_train_ratings.tsv*

The first contains a listing of book titles, and their corresponding IDs and URLs (or URIs actually) in DBpedia. We will parse this file and put it into an RDBMS, and for each URL, we will scrape the web contents and put them into an RDF triple that will be stored in 4store. Similarly, we will also get the data associated with each book from the list using the API of Goodreads and our local MongoDB will be populated with such data.

The second two datasets contain *user-item-rating* values which will be also stored in our local RDBMS. Accordingly, the main RDBMS will contain as much information as needed in order to be then able to link users that rate books (and the given rate), and to the data obtained from those two external sources already introduced.

These three datasets will form the initial basis for our system, such that a user who first enters the site will have data for these books/ratings immediately locally accessible.

Obtaining and Querying RDF data

The technology chosen for storing RDF data is 4store. The reason to choose 4store is that it perfectly fits both in this project statement and in our requirements, while other platforms such as Virtuoso are quite heavy and too hardware-dependent, or other possible solutions such as using a graph database lack of documentation. Oppositely, 4store is a light (native) triplestore, it is easy to install and to run on every machine and supports SPARQL queries. Moreover, and most important for us, it is already provided with Python libraries, which is the programming language chosen to implement the system.

The data stored in the local 4store will be mainly queried by means of SPARQL, so the respective project requirement is fully accomplished.

Obtaining and Querying XML / JSON data

For the JSON data source, MongoDB has been chosen. Basically, this is so because most of external sources we have chosen for "documented" data return JSON rather than XML, so in general lines we avoid impedance mismatch from translating from JSON to XML if we choose a JSON (native) database rather than an XML database. In addition, MongoDB has already made its place in the database world so there is a huge community providing support and there are already lots of tools out there to manage your MongoDB instances.

In this case, but similarly to the RDF data, the data will be retrieved from the Goodreads API. This is the source chosen to obtain the JSON data because it contains way more information than other APIs considered. Some examples of the APIs discarded are:

- ISBNDB:

http://isbndb.com/api/v2/json/OXKV6A0L/books?q=catcher in the rye

- Google APIs

https://www.googleapis.com/books/v1/volumes?q=the%20catcher%20in%20the%20rye+inauthor:salinger

- Library of Congress

http://lx2.loc.gov:210/lcdb?version=1.1&operation=searchRetrieve&query=catcher%20in%20the%20rye&maximumRecords=5

Nevertheless, some Goodreads APIs return the requested information in XML format instead of JSON. In such scenario, we will also need to include a small parser phase aimed at translating the result into JSON if needed, so it can be finally stored in MongoDB. An example of Goodreads API result is as follows:

http://www.goodreads.com/search.xml?q=the+catcher+in+the+rye%20&key=g1kKTmrhRCN8IpXI8elXQ

Obtaining data from APIs

The API requirement will be fulfilled by means of Goodreads as well. Our decision for this part consists of using mainly our 4store and MongoDB local stores, and let the API source (Goodreads API) for those cases when the desired data cannot be found in any of our local sources. We would also like to use DBpedia API, but since the information DBpedia provides through its API is very poor compared to its SPARQL endpoint, the system will access it by means of SPARQL as well.

Local Data Storages Architecture

Data Storage

Just as it is already stated in the previous introduction, in this project we bet for having data stored locally in our databases. This implies that these imported data must be up-to-date in order to answer queries with fresh data. Nevertheless, if we take into account that “external world data” cannot be fully trusted, then the problem of updating local data loses importance, since it can now be solved by means of scheduled update processes. Thus, we don't actually need to keep our local database up-to-date to the last millisecond, but it is enough to update it daily, weekly, or even monthly.

Local Stores Population

In this section, we discuss the population process used for both the JSON and RDF local stores.

JSON data – MongoDB

The process followed to store Goodreads data into our JSON storage system is to take the list of books from the file given in the project statement and to call the Goodreads API against each one to return an XML result which is then translated inline to JSON and stored locally. Afterwards, this process goes through the book list again and, for each corresponding JSON file retrieved from Goodreads, the best result of each is selected. In other words, the Goodreads API call returns all the possible results to match a given book, but the system needs to pick the best amongst those, which in our case is done by simply choosing the ones with direct hits on the title (and then choosing which has the most ratings). After the system has chosen the best return result for a given book, it imports that JSON file into our local MongoDB.

RDF data – 4store

In order to populate 4store, the capabilities of SPARQL are exploited and the **CONSTRUCT keyword** instead of the classical SELECT one is used. The advantage offered by this new keyword is that it provides with the possibility of defining triple patterns, so the data format given in return is already triple-like. Thus, we avoid impedance mismatch from translating SELECT query results into real triples, and therefore the overall performance of this process can be assumed to boost. Note that, since we are using a triplestore, the data we are passing to it as input must be in triple format.

Additionally we have to decide what data (i.e., what RDF properties) are going to be extracted from DBpedia. DBpedia is a very rich data source, but not everything it contains can be integrated with the rest of the sources. Since this populating process may also take place during the recommendation system (e.g., to retrieve books that are not locally stored in 4store), it has been designed to use two plain files containing the data to be obtained. That way, it can be updated any time by just modifying those files:

- The first one of them is what we call a **prefix file**. This file should contain the URI prefixes to be used in the SPARQL queries.
- The second file is called **data file** and it contains the actual properties to be retrieved from DBpedia.

The role of the RDBMS

We will initially have four tables: books, ratings and users. The **users** table will just be a table with IDs for the users (and if later we want more information, it can be easily extended). The **books** table will be (**ID, title**), where **ID** will be the ID that the row gets upon inserting an object. The **ratings** table will be **userID, bookID, ratings.**

In such scenario, one fast lookup to those relational tables will allow the system to detect whether the query currently to be solved is part of a cold start process or not, plus if the book is stored in our local sources. If that is the case, then the metadata rules inherited from the JSON config file (see below) will be very simple since the integration will be easy. Otherwise, if the book is not stored locally and therefore the system needs to look for that book in any external source, then the metadata needed will be more complex and probably based on probabilities.

Metadata Repository

JSON Metadata Repository

The JSON config file will contain the following entries:

- **Frontend:** It will contain what the frontend client is allowed to search.
- **External sources:** A list of the different sources the system is going to use. Each one of them will contain the subentries:
    - **Searchby:** This keyword points from a global to a local concept that can be used to search. For instance, the global concept title is translated to an URI referring to the corresponding resource in the case of DBpedia.
    - **Mapping:** This keyword is similar to the **search by** entry, but it is just to map between global and local concepts that cannot be used to search but are to be send back to the frontend in order for it to display them.
    - **Matching:** This keyword allows the middleware to identify what attributes are the same between sources so when it comes to undertake a matching process, it can actually be done through these attributes.
- **Backend:** Containing the same keywords and the same objective than the **external sources** but referring to the local RDBMS only.
- **Search rules:** Additional rules (e.g., boolean variables) to configure the system behavior in certain situations. It also includes a main rule that defines what attributes are considered to determine if two books are similar (e.g., the author and the literary genre).

This then implies that the **external sources** and **backend** will each provide how to search using those terms against their data. Note that we previously defined the RDBMS to be accessed prior to the JSON metadata file in terms of the current query, but this does not exclude the RDBMS to be accessed through metadata as well.

Entity resolution and Integration

Entity resolution for Goodreads

Three methods based on the title:

1. **Select a book based on direct hit**. This means the query directly matches the book title. For this, the punctuation is removed and the word *the* not counted. If there are multiple direct hits (i.e., the query is a substring of the book title), then we choose the result which has the most ratings.

2. **Select based on high confidence**. This method will essentially be used in the case that there is no **direct hit** for that book. It splits the query into words and then searches the results that Goodreads returns and finds if any contains over half the words in the

query. The *confidence threshold* which says whether to accept the book or not is based on the words counted.

3. **Select based on count ratings**. This is only called in the case that the first two do not return positive results, and it is dangerous because it can return false positives, but also smart in that it can occasionally return the right result which slipped past both the first two cases (which can be a little too restrictive at times)

Entity resolution for DBpedia

Similar to the Goodreads case, entity resolution in DBpedia is based on the combination of both **direct hit** and **high confidence** methods.

More specifically, given a book title the system needs to search for, it first splits the title into words and runs a global query combining those words in an ordered regular expression. Afterwards, a probabilistic matching function takes place, evaluating each of the results obtained. This function needs to be a little bit more sophisticated than in the Goodreads case, since no rating data is stored now to choose amongst books. Accordingly, the entity resolution process will evaluate the titles of candidate books by means of:

- Scoring each matching word between the original title and the candidate title.
- Penalizing those words that appear in the original title but not in the candidate title.

After computing each candidate book score, the system will sort them in a descend manner. Thus, books with higher scores will be those closer to the **direct hit** resolution, so this method will have priority (like in the Goodreads case). Penalizing candidate books will allow the system to discard books in case that the score they get is too low. The penalization a book obtaints will be proportional to the number of letters it misses with regard to the total number of words in the original title.

Integration

The identification of two pieces of data (one from Goodreads, and the other from DBpedia) will be done basically by means of comparing the title and the author. If the titles from both sources match (or one is a substring of the other or some combination like that), and the author obtained from Goodreads matches the author obtained from DBpedia (DBpedia provides the author name in many formats), then the system will conclude it has found the correct book. Otherwise, the system will return results from DBpedia.

Recommender

The recommender system is mainly based on evaluation of distances between registered users. The choice of a user-based system (compared to an item-based where the distances would be computed on books) is justified by two assumptions: First that there will be less users than books in our databases, so it will be less costly to evaluate distances between users. Second we will assume that those distances are quite stable and it won't be necessary to update them at every change in user profiles. For instance an update once a week would be sufficient. In our particular case though, as our user database is not too big, we can compute it at each recommendation.

The recommender has to handle different cases:

- A cold case: recommendation of Books when we have no information on preferences of a user, and when he is not looking for a particular book. This recommendation is based on a weighted average of ratings. The returned list will show all the books that are in our local Database, ranked by ratings (but giving more weight to books that have more reviews).
- A user-history based case: Given preferences of a user, we will return a list of other books he has not read and he might like. This is based on the preferences of other user similar to him.
- A Book-based case: If a user is searching for a book, we can return recommendation of similar books (without need for any user information). This will be done in two different ways. First a content-based similarity (books with same author, title, genre …). The second recommendation will be user-based (people who like this book, also like this one). The combination of those two lists will ensure diversity of results.

Then of course, any combination of those cases is also possible. For instance if we know a user and he is looking for a book, we can return a cross list using methods (2) and (3).

Finally the recommender should be able to adapt to binary ratings and non-binary ones. This will be done while computing distances. For binary ratings the metric used is the Euclidian metric. For non-binary ones, we prefer to use Pearson distance, as it gives similarity between users taking into account not just the numerical rating but more the global behavior (it will be able to compare a user that only gives ratings from 3 to 5 and another one who only gives 0/5 ratings if they like/disliked the same books).

References

- SEGARAN, Toby. *Programming collective intelligence: building smart web 2.0 applications*. " O'Reilly Media, Inc.", 2007.
- http://challenges.2014.eswc-conferences.org/index.php/RecSys, 19-03-2014