Content-based Recommender Systems

John Bruntse Larsen

Kongens Lyngby 2013
IMM-M.Sc.-2013-67

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-M.Sc.-2013-67

Summary (English)

The goal of the thesis is to evaluate content-based recommender systems in the domain of video games. The thesis compares approaches based on Linked Open Data and natural language processing(NLP) with traditional approaches which are only based on NLP methods.

Summary (Danish)

Målet med afhandlingen er at evaluere indholdsbaserede anbefalersystemer i domænet af computerspil. Afhandlingen sammenligner tiltag baseret på Linked Open Data og sprogteknologi(NLP) med traditionelle tiltag der udelukkende er baseret på NLP.

Preface

This thesis was prepared in spring 2013 at the department of Computer Science at Korea Advanced Institute of Science and Technology(KAIST), and finished in July 2013 at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Informatics.

The thesis deals with content-based recommender systems where most of my background in information retrieval is from a semester of courses at KAIST and projects at the Semantic Web Research Center at KAIST.

The thesis consists of this report and a web-application that demonstrates a content-based recommender system in the domain of video games.

Lyngby, 11-July-2013

John Bruntse Larsen

Acknowledgements

I would like to thank Professor Tony Veale and Dr. Martín Rezk for helpful discussions and feedback during during the early stages of the thesis. As external expert, Professor Tony Veale has also been very helpful with critique and suggestions during the thesis. I thank my co-supervisor at DTU Professor Jørgen Fischer Nillsson for helping with the schedule and clarifying the problem statement. I thank Professor Key-Sun Choi for being the primary supervisor of the thesis. I will also thank Professor Jørgen Villadsen at DTU for being my personal supervisor during the master degree. I would like to thank the students Kyung Tae Lim and Hahm Myoung Gyun for showing interest in the thesis and providing moral support.

Contents

Chapter 1

Introduction

As the speed of the Internet increases, it has become more viable to distribute software as digital downloads (commonly referred to as *digital distribution*). Digital distribution is one of the features of *Steam* made by the game developer *Valve*. In Steam, users created an account, bought games for the account through the Web and then downloaded both the games and future updates through a local client on a PC. Since its release in 2004, Steam has expanded as a distribution platform where any game developer can submit a game to be distributed digitally, under the supervision of Valve.

Recently Valve launched *Steam Greenlight* (from here simply referred to as *Greenlight*), where game developers can submit text and other media to a web page for their game. The Steam users can visit this page and vote for the the game to become available in Steam to be bought. Figure 1.1 shows a part of the submission for the game "Primordia" and the lifetime of a game in this system. As such, Greenlight provides a platform for developers to promote their unreleased products to potential customers.

Greenlight features a recommender system which presents only a few submissions to users, however in the interviews with game developers on Greenlight, it is criticized for favouring only the popular content. To address this problem, the recommender system could instead make recommendations based on the content of the games and products ratings for the given user. Figure 1.2 demonstrates

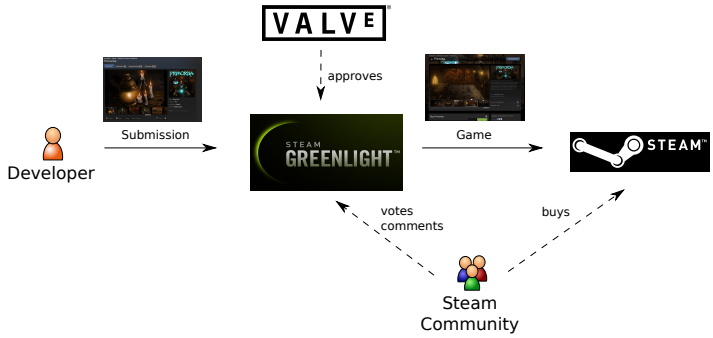


Figure 1.1: The lifetime of a game in Steam and Greenlight. The developer submits a description to Greenlight which can be rated and commented by the Steam users. Eventually Valve may approve of the game to be included in Steam where it can be bought.

such a recommendation with the Greenlight submission "Primordia" which will be used as a running example in the report. Because of the high correlation between the properties of a game likened by the user and the submission, the recommender system should both recommend "Primordia".

1.1 Recommender Systems

The purpose of a recommender system for an user is to present the most relevant products from a larger set of products. In the domain of this project the set of products are Greenlight submissions and the relevant submissions are those which a specific Steam user would want to discover, rate and eventually buy. Recommender systems can very roughly be categorized into two distinct approaches (*i*) *content-based* and (*ii*) *collaborative*. A third approach (*iii*) *hybrid* combines the other two approaches [AT05].

Content-based recommender systems finds the products that are similar to those that the user likes by measuring similarity between products. With this approach, a fitting product can be recommended to the user even though no one have rated it before. In [AT05] they following issues for content-based recommender systems

Limited Content Analysis Automatic profiling of products can be difficult and manual profiling can be impractical.


	Machinarium	Primordia
Genre/gameplay	Point-and-click adventure	Point-and-click adventure
Theme/setting	World of robots	World of robots
Theme/setting	Steam punk	Post-apocalyptic

Figure 1.2: Assumed characteristics of the Steam game "Machinarium" (left screenshot) and the Greenlight submission "Primordia" (right screenshot). Each rows state some property about the games. Assuming the user likes Machinarium, Primordia should be recommended due to similar characteristics.

Overspecialization The recommender system should attempt to achieve diversity among the recommendations and avoid recommending products that are too similar to the profile.

New User Problem New users without a profile do not get accurate recommendations. This is known as the cold-start problem for new users.

Collaborative recommender systems finds the products which the highest ratings from other users. A pure collaborative approach eliminates the cold-start problem for new users, however it introduces a cold-start problem for new products which do not have any ratings yet.

A hybrid recommender system is any recommender system which combines content-based and collaborative recommendations in one system. Hybrid recommender systems attempt to combine the best of content-based and collaborative approaches and in [Bur07] several different hybrid approaches are evaluated and compared.

Additional approaches presented in [Bur07] are *knowledge-based* systems, which are similar to content-based systems and rely on inferring the user needs, and *demographic* systems which groups users and products into demographic niches by using ratings of the users. Figure 1.3 from [Bur07] shows a comparison by the different approaches by their input.

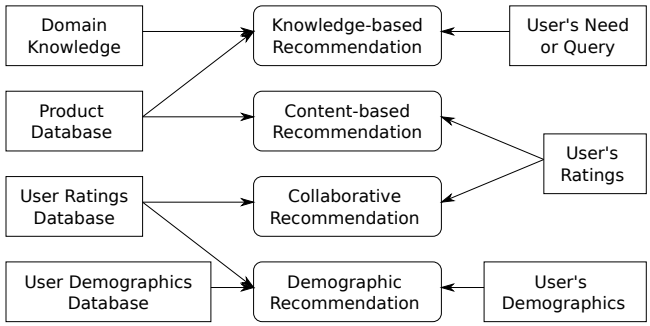


Figure 1.3: Input for different recommendation approaches.

The running example from figure 1.2 is of a content-based recommender system as the listed properties make out a product profile and the recommendation is based on a positive rating of the product "Machinarium".

The following sections introduce a few approaches for profiling products and measuring similarity, which addresses the limited content analysis issue.

1.2 Products as Documents

Typically products are described in documents of text intended to be read by humans and measuring similarity between such documents is a well-studied area in *Natural Language Processing* (NLP) with many available methods and tools. Relations between words are stated informally and the following methods attempt to measure similarity from such words. The following sentences will be used as an example to illustrate the methods although the actual corpus is generally much larger. The sentences are not related to the domain of this project.

Sentence 1: *Trees can be trees in nature or trees in computer science.*
Sentence 2: *It can be dangerous to be in caves.*
Sentence 3: *In nature, monkeys can live in trees or caves.*

1.2.1 Term-Document matrix

In information retrieval, the term frequency-inverse document frequency (TF-IDF) is a very common empirical metric to measure the significance of a word

	Trees	*can*	*be*	*in*	*nature*	*or*	*computer*	*science*
TF	$\frac{3}{11}$	$\frac{1}{11}$	$\frac{1}{11}$	$\frac{2}{11}$	$\frac{1}{11}$	$\frac{1}{11}$	$\frac{1}{11}$	$\frac{1}{11}$
IDF	$\log \frac{3}{2}$	0	$\log \frac{3}{2}$	0	$\log \frac{3}{2}$	$\log \frac{3}{2}$	$\log \frac{3}{1}$	$\log \frac{3}{1}$
TF-IDF	0.048	0	0.016	0	0.016	0.016	0.043	0.043

Table 1.1: Example of TF-IDF of the words in Sentence 1 with 9 words, where the letter-case is ignored. TF-IDF is the product of TF and IDF. According to TF-IDF *Trees* is the most relevant word in this document, followed by *computer* and *science*.

in a document. The TF-IDF is the product of the term frequency (TF) and the inverse document frequency (IDF) and is generally high for words which are uncommon among all documents but which occur frequently in a few documents. Table 1.1 shows the TF, IDF and the TF-IDF of the words in sentence 1 of the example. Notice that exact definition of a word is let open to the individual application and the measure is purely based on syntax. Often *stop-words* such as "in" which occur very frequently are considered insignificant and ignored when calculating TF-IDF. Notice that the word *can* is ambiguous and may not be considered a stop-word. The row vector of TF-IDF values can be considered a *document-vector* for sentence 1.

Given a set of documents, the *term-document* matrix is a matrix of document-vectors as columns. The rows in the term-document matrix are often referred to as *term-vectors*. Table 1.2 shows a term-document matrix for the example sentences with TF-IDF values. Both of the following methods compute similarity between documents by using a term-document matrix.

1.2.2 Simple Vector Correlation

In this method, similarity between documents is measured directly from the term-document matrix by correlating document-vectors with cosine similarity

$$sim(\overrightarrow{v}, \overrightarrow{u}) = \cos\theta = \frac{\overrightarrow{u} \cdot \overrightarrow{v}}{|\overrightarrow{u}| \cdot |\overrightarrow{v}|}$$

where θ is the angle between $\overrightarrow{v}$ and $\overrightarrow{u}$. With the example term-document matrix in figure 1.2, the similarity values for Sentence 1 with the other sentences are as such

$$\overrightarrow{Sent.1} \cdot \overrightarrow{Sent.2} = 0.016 \cdot 0.045 = 0.0007$$

$$sim(\overrightarrow{Sent.1}, \overrightarrow{Sent.2}) = \frac{0.0007}{0.081 \cdot 0.052} = 0.166$$

	Sent. 1	Sent. 2	Sent. 3
trees	0.048	0	0.020
can	0	0	0
be	0.016	0.045	0
in	0	0	0
nature	0.016	0	0.020
or	0.016	0	0.020
computer	0.043	0	0
science	0.043	0	0
it	0	0.06	0
dangerous	0	0.06	0
to	0	0.06	0
monkeys	0	0	0.053
live	0	0	0.053
caves	0	0.023	0.020

Table 1.2: The term-document matrix for the example with TF-IDF values.

$$\overrightarrow{Sent.1} \cdot \overrightarrow{Sent.3} = 0.048 \cdot 0.020 + 2 \cdot (0.016 \cdot 0.020) + 0.023 \cdot 0.020 = 0.002$$

$$sim(\overrightarrow{Sent.1}, \overrightarrow{Sent.3}) = \frac{0.002}{0.081 \cdot 0.083} = 0.297$$

According to this method, Sentence 1 and Sentence 3 are more similar than Sentence 1 and Sentence 2.

The major advantage of this method is that it is simple to implement and will in this project serve as the baseline similarity measure. A disadvantage of this method is that the document vectors are generally sparse and synonyms are not considered. It is also problematic to include new documents as it changes the IDF values of words, which can be a problem as Steam Greenlight is frequently updated.

1.2.3 Latent Semantic Indexing

Latent Semantic Indexing [DDF+90] (LSI) attempts to address some of the disadvantages of computing similarity directly from the term-document matrix by first applying Singular Value Decomposition (SVD) to the matrix. The underlying intuitive assumption of LSI is that words that are highly related often occur in the same documents and that this can be used to expand document queries. It is out of the scope of this project to cover the theoretical background of SVD in detail but in short it approximates a matrix A by decomposing it into

a product of three smaller matrices by computing eigenvalues and eigenvectors. The SVD of matrix A with k factors are written as follows.

$$A_k = S_k \Sigma_k U_k^T$$

In LSI, A_k is the term-document matrix, S_k are k-dimensional term vectors, U_k are k-dimensional document vectors and Σ_k are a diagonal matrix with the k largest *singular values*. As an example, the result of SVD applied to the term-document matrix in figure 1.2 for $k = 2$ are the following matrices.

$$\Sigma_2 = \begin{bmatrix} 0.61 & 0.0 \\ 0.0 & 0.12 \end{bmatrix}$$

$$S_2 = \begin{bmatrix} 0.08 & 0.01 \\ 0.00 & 0.00 \\ 0.03 & 0.39 \\ 0.00 & 0.00 \\ 0.03 & 0.01 \\ 0.03 & 0.01 \\ 0.70 & -0.01 \\ 0.70 & -0.01 \\ 0.00 & 0.52 \\ 0.00 & 0.52 \\ 0.00 & 0.52 \\ 0.00 & 0.03 \\ 0.00 & 0.03 \\ 0.00 & -0.21 \end{bmatrix} \begin{matrix} trees \\ can \\ be \\ in \\ nature \\ or \\ computer \\ science \\ it \\ dangerous \\ to \\ monkeys \\ live \\ caves \end{matrix}$$

$$U_2 = \begin{bmatrix} 1.00 & 0.002 \\ -0.002 & 1.00 \\ 0.004 & 0.07 \end{bmatrix} \begin{matrix} Sent.1 \\ Sent.2 \\ Sent.3 \end{matrix}$$

A query in LSI is a list of terms l, which is converted to a k-dimensional vector *query* by summing together the corresponding term vectors of S_k. As an example, suppose l is the list of terms in Sentence 1 which have a non-zero TF-IDF

$$l_{Sent.1} = [trees, be, nature, or, computer, science]$$

$$\overrightarrow{query_{Sent.1}} = \overset{trees}{\begin{pmatrix} 0.08 \\ 0.01 \end{pmatrix}} + \overset{be}{\begin{pmatrix} 0.03 \\ 0.39 \end{pmatrix}} + \overset{nature}{\begin{pmatrix} 0.03 \\ 0.01 \end{pmatrix}} + \overset{or}{\begin{pmatrix} 0.03 \\ 0.01 \end{pmatrix}} + \overset{computer}{\begin{pmatrix} 0.70 \\ -0.01 \end{pmatrix}} + \overset{science}{\begin{pmatrix} 0.70 \\ -0.01 \end{pmatrix}}$$

$$= \begin{pmatrix} 1.57 \\ 0.40 \end{pmatrix}$$

The correlation between *query* and a document vector from U_k can then be used for measuring document similarity. In the Lingpipe tutorial, the vectors are also

scaled by the corresponding eigenvalues. Using this approach for this example, the similarity values for Sentence 1 are as such

$$|\overrightarrow{query_{Sent.1}}| = \sqrt{1.57^2 \cdot 0.61 + 0.40^2 \cdot 0.12} = \sqrt{1.52}$$

$$\overrightarrow{query_{Sent.1}} \cdot \overrightarrow{Sent.2} = 0.61(1.57 \cdot (-0.002)) + 0.12(0.40 \cdot 1.00) = 0.046$$

$$sim(\overrightarrow{query_{Sent.1}}, \overrightarrow{Sent.2}) = \frac{0.046}{\sqrt{1.52} \cdot \sqrt{(-0.002)^2 \cdot 0.61 + 1.00^2 \cdot 0.12}} = 0.11$$

$$\overrightarrow{query_{Sent.1}} \cdot \overrightarrow{Sent.3} = 0.61(1.57 \cdot 0.004) + 0.12(0.40 \cdot 0.07) = 0.0072$$

$$sim(\overrightarrow{query_{Sent.1}}, \overrightarrow{Sent.3}) = \frac{0.0072}{\sqrt{1.52} \cdot \sqrt{0.004^2 \cdot 0.61 + 0.07^2 \cdot 0.12}} = 0.24$$

According to this method, Sentence 1 and Sentence 3 are more similar than Sentence 1 and Sentence 2. The same conclusion was found using the simple vector correlation approach in section 2.2.

A major advantage of LSI is that it to some extend considers synonyms and addresses the sparsity problem of the original term-document matrix. An implementation of SVD is available in Java with the proprietary library *Lingpipe*, which is freely available for research use with freely available data and software. There is also an online tutorial on how to use Lingpipe for LSI. A disadvantage of LSI is that it involves solving an often large polynomial which takes a long time and must be done whenever a new document is added to the term-document matrix.

1.3 Products as Linked Open Data Entities

In the methods introduced in section 1.2, products were considered as documents and similarity were measured from informally related words. This section introduces an approach for content-based recommendation [DNMO+12] using Linked Open Data, where products are considered logical entities with formally stated properties.

Linked Open Data [BHBL09] (LOD) is a recent approach in the Semantic Web community for storing and accessing data on the Web in a standardized and structured way. Data are defined with triples such as (`Steam`, `developer`, `Valve`), to express that "the developer of Steam is Valve", where `Steam` and `Valve` are *entities* and `developer` is a *relation* (sometimes called a *property*). Such triples are accessed using the query language *SPARQL* [BHBL09].

A major advantage of the LOD is that entities have an URI so that other datasets can make a reference to them. In particular, equality between two entities in different datasets can be formally declared by the `owl:sameAs` relation, which allows information about a single entity to be distributed over multiple datasets. Instances of the relation may be defined manually or automatically with tools such as LIMES [NA11].

1.3.1 LOD for Content-Based Recommendation

In [DNMO+12] the authors used the datasets from *Dbpedia* [BLK+09], *Freebase* and *LinkedMDB* for a content-based recommender system in the film domain. Using SPARQL, they identified films as LOD individuals in the datasets, which gave triples such as the ones below for "Rightous Kill" and "Heat"

$$(\texttt{Rightous Kill}, \texttt{starring}, \texttt{Al Pachino}), (\texttt{Rightous Kill}, \texttt{starring}, \texttt{Brian Dennehy})$$

$$(\texttt{Rightous Kill}, \texttt{starring}, \texttt{Robert de Niro})$$

$$(\texttt{Heat}, \texttt{starring}, \texttt{Al Pachino}), (\texttt{Heat}, \texttt{starring}, \texttt{Robert de Niro})$$

These triples were transformed to vectors as below

$$\overrightarrow{\text{Rightous Kill}_{starring}} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \quad \overrightarrow{\text{Heat}_{starring}} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \begin{matrix} \textit{for Brian Dennehy} \\ \textit{for Al Pachino} \\ \textit{for Robert de Niro} \end{matrix}$$

These vectors were then correlated by cosine similarity was to measure the similarity between the two films, regarding the *starring* relation, and the overall similarity measure was averaged over all relations.

The precision/recall of a content-based recommender system based on this method was evaluated to be higher than content-based recommender systems based on NLP tools. The proposed reason was that the structure in the LOD gave higher quality vectors, than vectors of extracted keywords from documents.

1.3.2 Mixing NLP with Linked Open Data

The approach in [DNMO+12] works when all products can be identified as LOD entities, however the Greenlight submissions in this domain are typically very new and only described on the Web with traditional media.

The approach in this project is to apply *entity disambiguation* to automatically profile Steam Greenlight submissions with LOD triples from textual descriptions. The Hellman NLP2RDF [HLA12] online demo of entity disambiguation with *Dbpedia Spotlight* [MJGSB11] can map some of the words in a given text to entities in Dbpedia. For example from the string "- Gorgeous post-apocalyptic setting" in the description of the Steam Greenlight submission for "Primordia", Hellman NLP2RDF finds the entity `http://dbpedia.org/resource/Apocalyptic_and_post-apocalyptic_fiction`. From this result, a triple like

(`Primordia`, `subject`, `Apocalyptic and post-apocalyptic fiction`)

can then be constructed and used in the similarity measure as shown earlier. Additionally taxonomies can be used to systematically discover implicit triples.

Entity disambiguation is still a new research area and the implementations are assumed to improve for higher precision/recall. The similarity measure will be evaluated against other current approaches for document-document similarity measures, such as Latent Semantic Indexing [DDF+90] and simple token-vector correlation.

The aimed functionality of the recommender system is to take as input a list of Steam game titles and then output a ranked list of Steam Greenlight submissions.

1.4 Practical Details

In this project, data are extracted from the web and processed by local programs. Rather than combining the extraction and processing into one program, the project will be split into many small programs and data are stored as XML files. Some of the major advantages of using XML files compared to using a relational database are

1. The data is plain text and easy to modify
2. It is not necessary to manage a separate server with a database
3. The data is easy to copy and move for testing purposes
4. Many common programming languages have libraries for writing and querying XML files

Appendix A shows an overview of the used XML tags and a demonstration of the usage.

1.5 Evaluation Methods

In this thesis I evaluate the quality of the similarity measure and the quality of the recommender results. The reason for evaluating the similarity measure separately is that the recommender system may try to achieve diversity rather than the high similarity, but this also requires a strong similarity measure. Both evaluations retrieves and handles test data automatically rather than being based on human inquiries.

The similarity measure returns a numeric value for the similarity between two games, hence a ranking of similarity can be produced for a fixed game. I evaluate the *average precision* of this ranking, where the relevant games are the Steam games classified as similar by humans. I use the freely available human classification at the website www.giantbomb.com, which is maintained by the community of users in the website. For example for the Steam Greenlight game "Primordia", there are six of the similar games which can be bought in Steam. I then compute the average precision of the similarity measure of "Primordia" profiled as a Steam Greenlight submission. These classifications are generally only available for released games, which is why I only consider the released games in Steam Greenlight for the evaluation.

The recommender system returns a ranking of Steam Greenlight submissions and I evaluate the ranking based on purchases of released Steam Greenlight games. I consider the users who bought the game to "like" the game and should be recommended it by the recommender systems, which allows me to evaluate the average precision. At the time of writing there are 12 released Steam Greenlight games.

1.6 Problem Definition

This project considers the problems in content-based recommender systems in the domain of Greenlight submissions and implementing a demo recommender system.

The problems that are addressed in the project are to

1. Profile products (Steam games and Greenlight submissions).
2. Measure similarity between products.
3. Extract user profiles.
4. Recommend the products (Greenlight submissions) for a given user profile.

1.7 Expected Results

I expect the structured data to be more precise than the extracted keywords, even though they are based on NLP-tool results. The taxonomies should allow the system to filter out large amounts of irrelevant data and incorrect results from the NLP-tools, which should increase the precision of the similarity measure significantly.

The results are limited by the coverage of the Steam games as LOD though and games with only generalized data (close to the root in the taxonomies) can lead to noise.

1.8 Report Structure

Chapter 2 presents the similarity measures based on NLP in detail and chapter 3 presents the similarity measure based on LOD and NLP in detail. Chapter 3 also presents the recommender system which apply the LOD-based similarity measure. Chapter 4 presents the evaluation of the presented methods with results and discussion. Chapter 5 concludes the report.

Chapter 2

Pure NLP Approaches

In this project, the similarity measure based on Linked Open Data is compared with similarity measures based on keywords extracted from documents with more conventional methods. Here the items are profiled from their documents with keywords in a term-document matrix as introduced in section 1.2.1. The first method is by simple vector correlation using the term-document matrix directly as introduced in section 1.2.2 and the second method is by using Latent Semantic Indexing (LSI) as introduced in section 1.2.3.

Assuming that the games and their descriptions are already available, the process of computing similarity with keywords in both methods can roughly be split into the following steps

1. Construct term-document matrix as presented in chapter 1 from item documents
2. Compute similarity between items using the term-document matrix.

Section 2.1 shows the algorithms for computing the matrix in steps 1 which is used in both similarity measures. Step 2 is then done by using either simple vector correlation or LSI. Section 2.2 shows the algorithms for the simple vector correlation similarity measure and section 2.3 shows the algorithms for the LSI based similarity measure.

Variable Type	Definition
`num`	number
`id`	string
`doc`	string
`token`	string
`profile`	`token => num`

Function Name	Input Type	Output Type
`stopWords()`	N/A	`token list`
`tokenFrequncies(d)`	`doc`	`profile`
`tfVectors(ds)`	`id => doc`	`id => profile`
`dfs(tfVs)`	`id => profile`	`token => num`
`tfidfVectors(ds)`	`id => doc`	`id => profile`

Table 2.1: The variable and function type specification for computing the term-document matrix with TF-IDF values.

2.1 Term-Document Matrix

As introduced in 1.2.1, the value at position (i, j) in a term-document matrix is non-zero if term i is present in document j and zero otherwise. In this project the term-document matrix contains TF-IDF values as defined in section 1.2.1. The term-document matrix can be computed offline from the similarity measure but when using TF-IDF, the matrix must be updated whenever a new item is added.

The term-document matrix is computed in a functional style by using several smaller functions with input and output but no side-effects. PHP is not a very strongly typed language, but the functions are implemented according to the type specifications shown in table 2.1.

The overall approach to computing the TF-IDF term-document matrix is to first compute the TF term-document matrix and then multiply each element with the corresponding IDF value. The function `stopWords()` returns the list of words considered as stop-words and not tokens. The function `tokenFrequencies(d)` tokenizes a document `d` to a vector with the number of occurrences of each token in `d`. Tokens that occur in the list returned by `stopwords()` are ignored. The function `tfVectors(ds)` shown in figure 2.1 computes the TF term-document matrix for the list of documents `ds`. The function `dfs(tfVs)` shown in figure 2.2 computes the document frequency of each token that occur in the term frequency vectors `tfVs`. The function `tfidfVectors(ds)` shown in figure 2.3 computes then computes the TF-IDF term-document matrix for the documents `ds` where only the non-zero elements explicitly stated.

```
function tfVectors(ds (id => doc)){
  let tfVs := empty dictionary
    foreach (id,d) in ds{
    let tokenVector := tokenFrequencies(d)
      let l := sum(tokenVector)
      foreach (token,v) in tokenVector
        tokenVector[token] := v/l
      tfVs[id] := tokenVector
    }
  return tfVs
}
```

Figure 2.1: This function computes the term-document matrix with TF values for the given documents `ds`. A document `d` is tokenized by `tokenFrequencies(d)` by to a token vector with the number of occurrences of the token in the document. The term-vector of `d` is then achieved by dividing each element with the sum of all occurrences in the token vector.

```
function dfs(tfVs (id => profile)){
  let dfs := empty dictionary
  foreach (_,tokenVector) in tfVs{
    foreach (token,_) in tokenVector{
      let u := dfs[token]
      if (u = null) then
        dfs[token] := 1
      else
        dfs[token] := u+1
    }
  }
  return dfs
}
```

Figure 2.2: The document frequency of each term in `tfVs` is computed by iterating over all vectors and counting the number of occurrences. The underscore character `_` denotes an anonymous variable.

```
function tfidfVectors (ds (id => doc)){
  let tfs := tfVectors(ds)
  let dfs := dfs(tfs)
  let D := #ds
  let tfidfVectors := empty dictionary
  foreach (id,termVector) in tfVs{
    let tfidfVector := empty dictionary
    foreach (token,tf) in termVector{
      let c := dfs[token]
      let df := log (D / c)
      tfidfVector[token] := tf*df
    }
    tfidfVectors[id] := tfidfVector
  }
  return tfidfVectors
}
```

Figure 2.3: The term-document matrix with TF-IDF values is computed by first-computing the term-document matrix with TF values and then multiplying each element with the IDF value for the corresponding term.

For convenience, the Steam games and Greenlight submissions are stored in separate term-document matrices. Figure 2.4 shows an example of how a column in the Steam term-document matrix is stored as XML.

2.2 Simple Vector Correlation

In simple vector correlation, the column vectors of the term-document matrix are used directly for computing $sim(\overrightarrow{u}, \overrightarrow{v})$ as introduced in 1.2.2.

Given the term-document matrices in figure 2.5 as input, it is straightforward to implement the similarity measure in PHP using the functions specified in table 2.2. The function `sim(u,v)` shown in 2.6 computes the $sim(\overrightarrow{u}, \overrightarrow{v})$ for two given profiles `u` and `v`. The function `topSim(title,steam)` shown in 2.7 computes the similarity ranking for a given Greenlight submission `title` against all game profiles in `steam`.

Steam	Machinarium ...
machinarium	0.156
award-winning	0.023
⋮	⋮

```
<steam>
  <game>
    <uri>http://store.steampowered.com/app/40700/?snr=1
        _7_7_230_150_36</uri>
    <title>Machinarium</title>
    <description>About the...</description>
    <keyword word="machinarium" value="0.156"/>
    <keyword word="award-winning" value="0.023"/>
    ...
  </game>
  ...
</steam>
```

Figure 2.4: The term-document matrices with TF-IDF values are stored as XML files. Each `<game>` element corresponds to a column vector in the matrix, with only the non-zero elements explicitly stated.

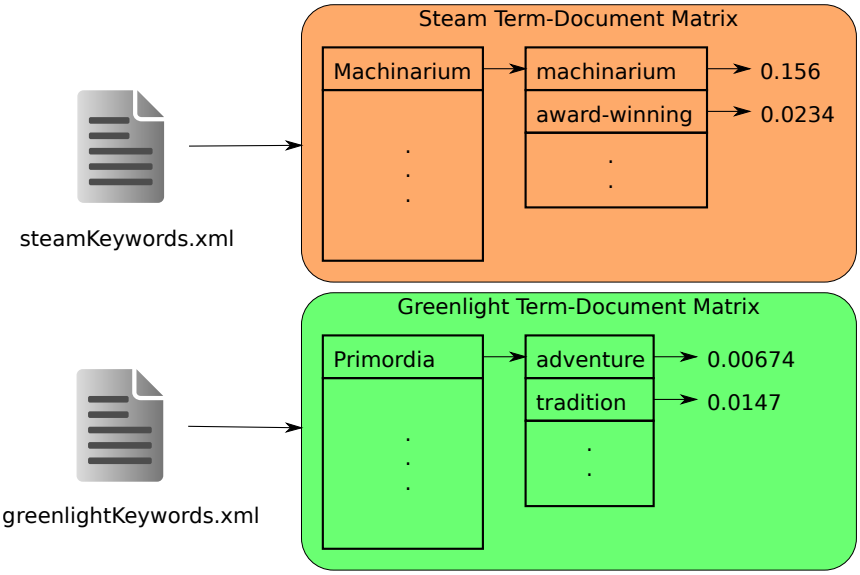


Figure 2.5: Input for the simple vector correlation. Each XML file defines a term-document matrix which is loaded into a separate dictionary in PHP as shown. The datatype for each dictionary is `id => profile` where a `profile` is a column vector of a term-document matrix.

Function Name	Input Type	Output Type
`sim(u,v)`	`(profile,profile)`	`num`
`topSim(title,steam)`	`(profile,id=>profile)`	`id priority queue`

Table 2.2: The function type specification for the simple vector correlation similarity measure.

```
function sim(u (profile), v (profile)){
  let vDu := 0
  foreach (word,x) in u{
    let y := v[word]
    if (y != null)
      vDu := vDu + x*y
  }
  return vDu / (sqrt(sum(u)) * sqrt(sum(v)))
}
```

Figure 2.6: Vectors are correlated by their cosine similarity. The returned value is always positive as there are no negative TF-IDF values.

```
function topSim(title (profile), steam (id=>profile)){
  let ranking := empty priority queue
  foreach (steamTitle,steamProfile) in steam{
    let v := sim(title,steamProfile)
    ranking := insert(steamTitle,v,ranking)
  }
  return ranking
}
```

Figure 2.7: For the evaluation, the games in `steam` are stored in a priority queue ordered by their similarity to `title`.

2.3 Latent Semantic Indexing

As introduced in section 1.2.3, Latent Semantic Indexing also uses the term-document matrix and cosine similarity but rather than using the column vectors directly, the matrix A is pre-processed by Singular Value Decomposition (SVD) which approximates it into a product A_k of three smaller matrices S_k, Σ and U_k^T.

$$A_k = S_k \Sigma_k U_k^T$$

The row vectors in U_k^T are then the document profiles which can be correlated by cosine similarity.

Rather than implementing SVD from scratch I use the Java based Lingpipe library which also has an online tutorial with source code for how to apply the library for an LSI based term-document similarity measure. In the tutorial program SVD is applied to a demo term-document matrix and given a list of terms as program arguments it will compute the query vector for those terms as shown in section 1.2.3 and print the results for that query.

In this project, the tutorial source code is wrapped inside the class `Lsi` specified in table 2.3. There are no major changes to the source code from the tutorial other than the results are returned rather than just printed. The `Lsi` class is instantiated with a given term-document matrix which is then processed by SVD in Lingpipe. After instantiation, the class can be queried with `query(terms)` where `terms` is a list of terms to get a dictionary with the document similarities for that query. The output dictionary typically looks like below

$$\texttt{query}\left(\begin{bmatrix}\text{adventure}\\ \text{tradition}\\ \vdots\end{bmatrix}\right) = \begin{bmatrix}\text{Machinarium} & \mapsto & 0.5\\ \text{Aura: Fate of Ages} & \mapsto & 0.2\\ \text{Half-life} & \mapsto & 0.15\\ & \vdots & \end{bmatrix}$$

Given the data shown in figure 2.8 as input, the program in the class `steamSimilarity` instantiates the `Lsi` class with the Steam term-document matrix and then uses the Greenlight term vectors as input for `query(terms)`.

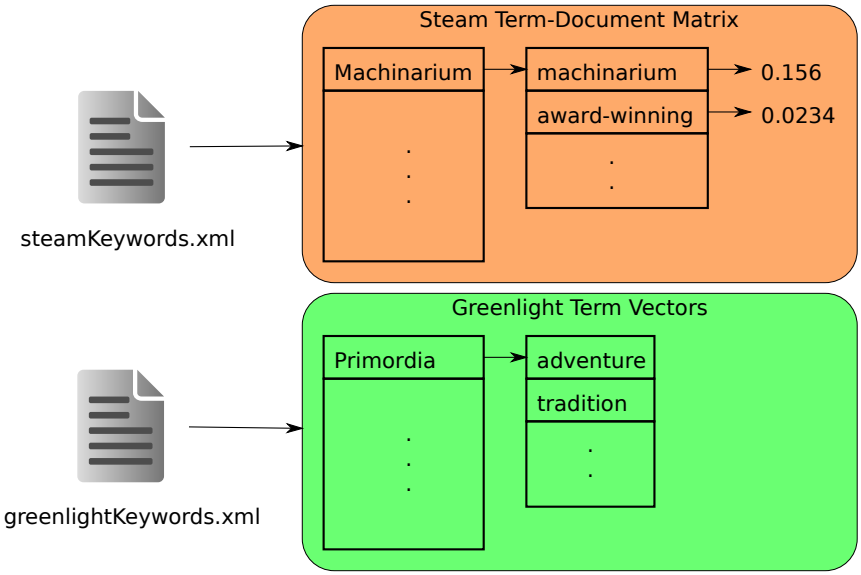


Figure 2.8: Input for the latent semantic indexing. The Steam term-document matrix is loaded as in section 2.2 and is processed with SVD using the Lingpipe library. The columns in the Greenlight term-document matrix are used as query vectors.

Lsi	
Type	Name
`double[][]`	`TERM_DOCUMENT_MATRIX`
`String[]`	`TERMS`
`String[]`	`DOCS`
`SvdMatrix`	`matrix`
`double[]`	`scales`
`double[][]`	`termVectors`
`double[][]`	`docVectors`
	`Lsi(String[] terms, double[][] tdm, String[] docs)`
`Map<String,Double>`	`query(String[] terms)`

Table 2.3: Specification of Lsi class for term-document similarity measuring. The constructor applies SVD to the input matrix using the Lingpipe library class `SvdMatrix`. The `SvdMatrix` class constructor parameters are not shown here and they are generally tuned to a particular case using empirical methods.

Chapter 3

Linked Open Data Approach

In the previous chapters, the similarity measure was based on extracting important keywords from documents using TF-IDF. It was also seen that words like "Adventure" which are quite common get a low TF-IDF score even though they are quite relevant to describing the game. Rather than using TF-IDF, *entity disambiguation* is used to identify words from documents in the LOD and the taxonomies are used to specify a vocabulary for the similarity measure. As presented in [DNMO+12], one of the advantages of using such structured data rather than just keywords are that the taxonomies can be used to find indirect links between entities.

Assuming that the Steam games and Greenlight submissions are already available, the process of computing similarity using with LOD can roughly be split into the following steps

1. Profile Steam games with LOD
2. Profile Greenlight submissions with LOD
3. Automatically optimize the profiles using the taxonomies of the LOD
4. Compute similarity using the algorithms and formulas in [DNMO+12]

The data in step 1 and step 2 will be referred to as *raw* data, since there has been no attempt at filtering out noise or other improvements at that point. The challenge in these steps is to *extract* LOD data using the available semantic web applications. The challenge in step 3 is to *analyse* the raw data and automatically optimize it so that it can be used in step 4 with no additional processing. Thus step 1-3 is preprocessing and only step 4 is used online by the recommender system.

The implementation is in PHP as it by default provides a big library of functions for extracting data from the web and handling of XML documents. To simplify the implementation in this project, a small custom library was written to wrap around the most commonly used functions. Despite PHP not being a functional language, the code is written in a functional style in that there are no mutating objects or other side effects and as such the functions are defined according to a given type specification. The functional style makes it easy to debug the code by testing the functions individually.

In Section 3.1, the data in Dbpedia is analysed in order to find a fitting way of handling and storing the data in the implementation. Section 3.2 shows how the raw data in step 1 and 2 are extracted and represented in PHP. In section 3.3, the raw data is analysed in order to identify implement a pipeline which automatically optimizes the data in step 3 and section 3.4 presents the implementation of step 4 and the recommender system which applies the similarity measure.

3.1 Analysis of Dbpedia Data

The triple space of RDF data is commonly illustrated as a three dimensional matrix, where each two-dimensional slice corresponded to a single property, with triple subjects in the rows and triple objects in the columns. The value at (i_s, j_o, k_p) is a positive real number if the triple (s, p, o) occurs in the dataset and zero otherwise. In [DNMO+12] they considered a subset of the Dbpedia triple space defined by a set of properties as illustrated in figure 3.1 with TF-IDF based values.

A column vector j in a slice matrix p is then the profile of the corresponding film m_j with respect to the property p. Rather than specifying the full column vectors, which are often very large due to sparse slice matrices, the vectors can be specified in terms of only the non-zero elements.

$$\text{Kill\_Bill} \overset{\texttt{dcterms:subject}}{\Rightarrow} \begin{pmatrix}\texttt{KillBill}\end{pmatrix} \quad \text{Kill\_Bill} \overset{genre}{\Rightarrow} \begin{pmatrix}\texttt{Fighting}\\ \texttt{Slasher}\end{pmatrix}$$

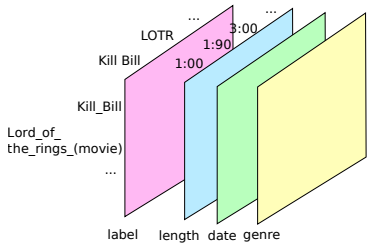


Figure 3.1: Triple space of film related RDF. Each slice is a two-dimensional matrix with triples for the corresponding property. The value at a given cell is zero if the triple do not exist in the dataset and non-zero if it exists in the dataset.

The entities in the `dcterms:subject` vector for an entity e corresponds to the Wikipedia Categories of the Wikipedia article for e. For example does *Kill Bill* have the Wikipedia article `http://wikipedia.org/wiki/Kill_Bill` and this article is listed in the single category `Kill Bill`. This is expressed in Dbpedia as the triple

```
dbpedia:Kill_Bill dcterms:subject category:Kill_Bill
```

The Wikipedia Categories are structured as a directed acyclic graph which forms a hierarchy of subcategories. For example is the category `Kill Bill` a subcategory of `Martial art films` and `Films directed by Quentin Tarantino` and has the single subcategory `Kill Bill characters`. In Dbpedia, the Wikipedia Category hierarchy has been mapped to the taxonomy defined by the `skos:broader` relation. For example is the above expressed in Dbpedia as the triples

```
category:Kill_Bill skos:broader category:Martial_arts_films

category:Kill_Bill skos:broader category:Films_directed_by_...

category:Kill_Bill_characters skos:broader category:Kill_Bill
```

In [DNMO+12], the authors increased the precision of the similarity measure by extending the `dcterms:subject` vector with the conclusions of the rule.

$$\frac{a \texttt{ dcterms:subject } b \quad b \texttt{ skos:broader } c}{a \texttt{ dcterms:subject } c}$$

The rationale of this approach is that the taxonomy can be used to perform a limited form of reasoning to add additional facts. Note that the `skos:broader` relations cannot simply be replaced by classic subsumption both because it is a relation between individuals and not classes and also because the taxonomy

is only based on a vague categorization. The YAGO ontology [SKW07] is an attempt to automatically build a proper subsumption-taxonomy based on the Wikipedia Categories but at a glance, it suffers from a few issues regarding game related data.

1. Outdated data access points

2. Shallow genre taxonomy

Under the assumption that Steam games exists as Dbpedia entities, the method used for films in [DNMO+12] can also be applied for the games to get vectors such as.

$$\texttt{Machinarium} \overset{\texttt{dcterms:subject}}{\Rightarrow} \begin{pmatrix} \texttt{Point-and-click adventure games} \\ \texttt{Adventure games} \\ \vdots \end{pmatrix}$$

$$\texttt{Machinarium} \overset{genre}{\Rightarrow} (\texttt{Graphic adventure game})$$

In this project the Steam games are compared with Greenlight submissions profiled by *entity disambiguation*. The problem in entity disambiguation is to annotate words in a document with suitable LOD entities. The output of the entity disambiguation is typically a list of triples as below

```
[adventure^^string scms:means dbpedia:Adventure_game, ...]
```

This project uses the Hellmann NLP2RDF [HLA12] web demo with the popular entity disambiguation tool Dbpedia Spotlight [MJGSB11].

Notice that simply disambiguating entities from the text document is not sufficient for applying the similarity measure with LOD as it is also necessary to identify the relation with the disambiguated entity. As a simple approximation, the relation is assumed to be `dcterms:subject` and the triples used for the profile will be the categories of the disambiguated entities. For example, if the entity disambiguation for the Greenlight submission *Primordia* found `dbpedia:Adventure_game` which have the following categories in Dbpedia

```
dbpedia:Adventure_game dcterms:subject category:Adventure_games
dbpedia:Adventure_game dcterms:subject category:Video_game_terminology
dbpedia:Adventure_game dcterms:subject category:Video_game_genres
```

```
<steam>
  <game>
    <title>Machinarium</title>
    <subject>Category:Point-and-click adventure games</subject>
    <subject>Category:Adventure games</subject>
    ...
  </game>
  ...
</steam>
```

Figure 3.2: Each `<game>` element defines the profile vector of one game. The title of the game is defined in the `<title>` element and the `dcterms:subject` related entities are defined in the `<subject>` element. The other slices are not considered since the Greenlight submissions only are profiled by the `dcterms:subject` relation. The root element shows if the profiles are for Steam games (`<steam>`) or Greenlight submissions(`<greenlight>`).

then the *Primordia* LOD profile will have these triples

`Primordia dcterms:subject category:Adventure_games`

`Primordia dcterms:subject category:Video_game_terminology`

`Primordia dcterms:subject category:Video_game_genres`

With respect to the triple space shown in figure 3.1, the Greenlight submissions will then only have values in the `dcterms:subject` slice with vectors like below

$$\text{Primordia} \overset{\texttt{dcterms:subject}}{\Rightarrow} \begin{pmatrix} \texttt{Adventure games} \\ \texttt{Video game terminology} \\ \texttt{Video game genres} \\ \vdots \end{pmatrix}$$

Storing these vectors separately allows the data extraction, optimization and similarity measure separately which is useful both for debugging purposes and reduces the computation time of the individual parts. For this project, the vectors are stored as XML files using a custom scheme as shown in figure 3.2. For larger applications it may be more suitable to use relational databases.

It is then simple to represent the vectors in PHP from the XML-files as dictionaries as shown in figure 3.3 using the default libraries. The dictionaries gives quick access to the profile for any given Steam game or Greenlight submission given its title. Following the functional style of implementation, the types used

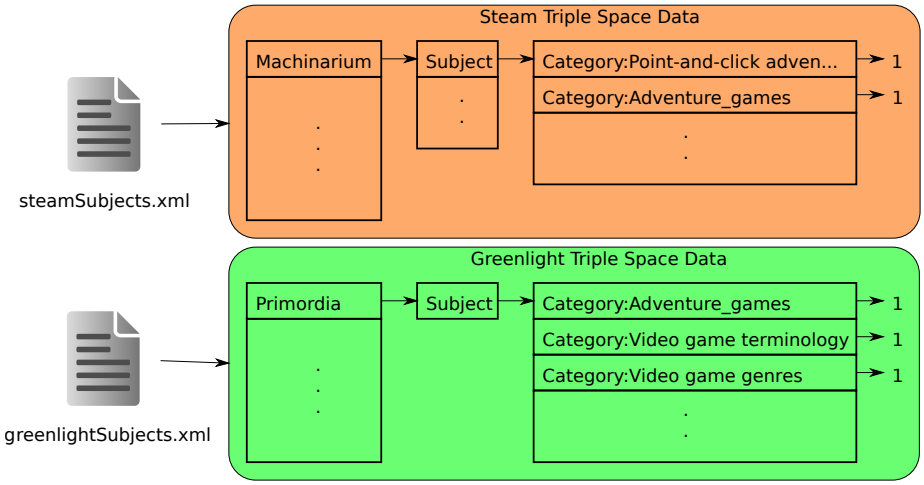


Figure 3.3: The triple data vectors are restored from the XML files in PHP as dictionaries. Although the Steam games are only profiled in terms of the slice defined by the `dcterms:subject` relation, the data structure can also handle vectors from any slice. Each mapped entity is mapped to a value, which in this case is always assumed to be 1, for calculating cosine similarity.

Type	Definition
`DOM`	`Document Object Model`
`id`	`string`
`entity`	`string`
`prop`	`string`
`num`	number

Table 3.1: Types used in the LOD similarity measure.

Function Name	Input Type	Output Type
`xmlAsDom(xml)`	`string`	`DOM`
`dbpediaSparql(query)`	`string`	`DOM`
`nlp2rdfResults(document)`	`string`	`DOM`
`dbpediaLinks(rdf)`	`DOM`	`entity` list

Table 3.2: Specification of PHP functions for answering SPARQL queries with Dbpedia. `xmlAsDom(xml)` parses an XML formatted string to a DOM object which can modified and queried in PHP. `dbpediaSparql(query)` returns a DOM object with the result of the given SPARQL query.

in this approach are specified in table 3.1.

In [DNMO+12] the authors additionally used the `owl:sameAs` relation to also include the triples from Freebase and the film dataset LinkedMDB to get additional information about each film. Although interlinking of datasets is one of the benefits of Linked Open Data, it is not in the scope of this project to evaluate with combined datasets.

3.1.1 Extracting Data from the Semantic Web

In the shown methods triples are extracted from Dbpedia and the result of the entity disambiguation are extracted from the Hellmann NLP2RDF web demo. In this project the PHP functions shown in table 3.2 are written to perform this extraction in an automatic and comprehensive manner. These PHP functions only requires the standard libraries of PHP.

In Dbpedia, triples can be accessed with the semantic web query language SPARQL and the Dbpedia SPARQL endpoint by using the following HTTP request

```
http://dbpedia.org/sparql?
default-graph-uri=http://dbpedia.org&
query=query&
format=format
```

Here *query* is the SPARQL query encoded for an URL and *format* is a constant specifying the format of the output (by default `text/html`). This request is wrapped into the PHP function `dbpediaSparql(query)` which returns a DOM

object with the result of the given SPARQL query `query` formatted as XML which is par.

The Hellman NLP2RDF web demo is typically accessed with a web browser where the user inputs a document, configures the application and press a button to get the result in the interface encoded as RDF. In the configuration, the application can be set up to use Dbpedia Spotlight for disambiguating the results to Dbpedia entities. Alternatively the application with only the Dbpedia Spotlight can be accessed with the following HTTP request

```
http://nlp2rdf.lod2.eu/demo/NIFDBpediaSpotlight?
input-type=text&
nif=true&
input=document
```

This request is wrapped into the PHP function `nlp2rdfResults(document)` which returns a DOM object with the results for the given document `document` formatted as RDF. The RDF output typically looks like in figure 3.4 where the disambiguated entities are specified with the `<scms:means>` tag. These entities are extracted into a list with the PHP function `dbpediaLinks(rdf)` where `rdf` is a DOM with the output from the HTTP request.

3.2 Profiling with Raw Data

The data that is extracted from the Semantic Web shown in section 3.1 is in this project referred to as *raw* data in the sense that it has not been optimized in any way. The raw data is expected to contain significant noise, both due to the informal background of the category taxonomy and because of incorrectly disambiguated entities. Nevertheless the *extraction* of this data are the first and second step in computing similarity between Steam games and Greenlight submissions with the LOD.

While it is a challenge to automatically identify the Steam games as Dbpedia entities, it only needs to be done once for each game. In this section, it is assumed that the Dbpedia entities for the Steam games are written in XML as shown in figure 3.5. It is then simple to query for the categories of the entity `uri` with the SPARQL query.

```
SELECT ?C WHERE <uri> dcterms:subject ?C
```

```
<rdf:RDF>
  ...
  <rdf:Description rdf:about="http://nlp2rdf.lod2.eu/nif/
      offset_3_12_adventure">
    <rdf:type rdf:resource="http://nlp2rdf.lod2.eu/schema/string/
        OffsetBasedString"/>
    <rdf:type rdf:resource="http://dbpedia.org/ontology/
        TopicalConcept"/>
    <str:anchorOf>adventure</str:anchorOf>
    <scms:means rdf:resource="http://dbpedia.org/resource/
        Adventure_game"/>
    <str:beginIndex>3</str:beginIndex>
    <str:endIndex>12</str:endIndex>
    <spotlight:support>958</spotlight:support>
    <spotlight:surfaceForm>adventure</spotlight:surfaceForm>
    <spotlight:offset>3</spotlight:offset>
    <spotlight:similarityScore>0.09329799562692642</
        spotlight:similarityScore>
    <spotlight:percentageOfSecondRank>-1.0</
        spotlight:percentageOfSecondRank>
    <spotlight:URI>http://dbpedia.org/resource/Adventure_game</
        spotlight:URI>
    <spotlight:types>
      Freebase:/media_common/media_genre,Freebase:/media_common,
          Freebase:/cvg/cvg_genre, Freebase:/cvg,Freebase:/award/
          award_discipline,Freebase:/award,DBpedia:TopicalConcept
    </spotlight:types>
  </rdf:Description>
  ...
</rdf:RDF>
```

Figure 3.4: Output of Hellman NLP2RDF with Dbpedia Spotlight plugin. The first `<rdf:Description>` element (not shown here) summarizes the input and the disambiguated substrings. Every other `<rdf:Description>` element describes a disambiguated substring, which is in the shown case for the word "adventure" starting at index 3 and ending at index 12. The `scms:means` element shows that the word was disambiguated as the entity http://dbpedia.org/resource/Adventure_game and the `spotlight` prefixed tags show some information regarding Dbpedia Spotlight.

```
<?xml version="1.0" encoding="utf-8"?>
<steam>
  <game>
    <uri>http://store.steampowered.com/app/40700/?snr=1
        _7_7_230_150_36</uri>
    <title>Machinarium</title>
    <description>About the ... </description>
    <dbpediaURI>http://dbpedia.org/resource/Machinarium</dbpediaURI
        >
  </game>
  ...
</steam>
```

Figure 3.5: Snipped of `steamDbpedia.xml` showing how the Steam game "Machinarium" is listed with a URI in Dbpedia

The PHP function `subjects(uri)` performs this query with `dbpediaSparql(query)` and returns the list of category entities which are then stored as shown in figure 3.2.

Similarly the Greenlight submissions are profiled with such vectors but they must be constructed from human-readable text. For this purpose, the text is parsed with Hellman NLP2RDF demo with the Dbpedia Spotlight entity disambiguation plugin which maps the words in the text to corresponding entities in Dbpedia. For example, from the text "An adventure in the tradition of Beneath a Steel Sky" in the submission of "Primordia" the result includes the mapping

$$\text{adventure} \Rightarrow \texttt{http://dbpedia.org/resource/Adventure\_game}$$

This entity will then have `dcterms:subject` relations into categories, such as `http://dbpedia.org/resource/Category:Adventure_games` which are used as values in the profile vector for the submission.

$$\text{Primordia} \overset{subject}{\Rightarrow} \begin{pmatrix} \texttt{http://dbpedia.org/resource/Category:Adventure\_games} \\ \vdots \end{pmatrix}$$

Summing up, the raw data for the Greenlight submissions are achieved as follows

1. Extract the description of Greenlight submissions
2. Map each submission to the output of the entity disambiguation
3. Query the Dbpedia sparql endpoint for subjects of each mapped entity.

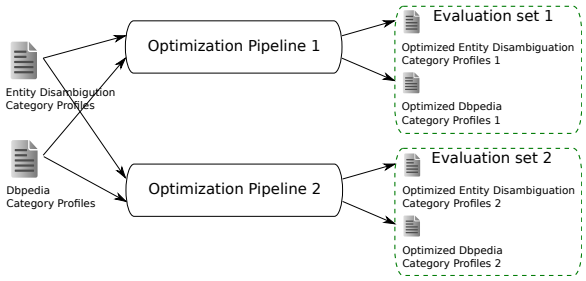


Figure 3.6: The category vectors extracted from the web and constructed from the entity disambiguation are post-processed by a pipeline of optimization modules in order to reduce the sparsity of the vectors. The format of the output is the same as the input.

3.3 Data Optimization

With the data from section 3.2 it is now possible to compute similarity between Dbpedia entities and the documents processed with entity disambiguation. However the high sparsity in the category vectors remains a problem. This was less of an issue in [DNMO+12] where other property vectors were included as well, but when only considering one property the similarity measure can be expected to perform poorly. This section presents methods for improving the performance by post-processing the category vectors in pipelines of optimization modules as shown in figure 3.6. Each module is designed to work as in individual program which takes some files with category vectors as input, process it and write the resulting category vectors in a new file as output. The benefit of this design is that each module can be tested independently and that it is simple to plug and unplug modules to get different pipelines.

3.3.1 Category Intersection Modules

The category vectors often contain categories which do not reflect content but rather some meta-data. As an example, the category `Cancelled Xbox 360 games` only states some historical information and is not useful in content-based recommendation. By removing such categories the vector cardinality is reduced and may also lower the vector sparsity. The problem is then to define the categories that reflects the content which depend on the domain. Since the two sets of category vectors are made from very different sources, it is also appropriate to consider them separately.

The category vectors for the Steam games are defined manually by humans and are hence assumed to generally be "correct". The categories are often subcategories of the `video games` category which branches out into widely different thematic topics. For example for "Machinarium" there are many categories which only state some meta-information about the game, such as released platforms and dropped platforms, rather than the content of the game.

$$\text{Machinarium} \overset{\texttt{dcterms:subject}}{\Rightarrow} \begin{pmatrix} \texttt{Point-and-click adventure games} \\ \texttt{Adventure games} \\ \texttt{Flash games} \\ \texttt{Windows games} \\ \texttt{Linux games} \\ \texttt{Indie video games} \\ \texttt{Cancelled Xbox 360 games} \\ \texttt{Video games developed in the Czech Republic} \\ \texttt{Art games} \\ \texttt{Amanita Design games} \\ \texttt{Mac OS X games} \\ \texttt{Steampunk video games} \\ \texttt{Cancelled Wii games} \\ \texttt{2009 video games} \\ \texttt{Independent Game Festival winners} \\ \texttt{Humble Indie Bundle games} \end{pmatrix}$$

The goal is to filter out categories which are not related to the genre or the theme of the game.

$$\text{Machinarium} \overset{\texttt{dcterms:subject}}{\Rightarrow} \begin{pmatrix} \texttt{Point-and-click adventure games} \\ \texttt{Adventure games} \\ \texttt{Art games} \\ \texttt{Steampunk video games} \end{pmatrix}$$

By inspecting the `skos:broader` hierarchy shown in figure 3.7, the categories related to genres and themes are found to have the lowest common ancestor `video games by genre`. This may seem incorrect depending on the interpretation of a genre and a theme however this is a consequence of the informal thematic hierarchy.

The categories are extracted from Dbpedia with the simple recursive depth-first graph traversal from a given start node shown in figure 3.8. The traversal returns a dictionary with the category names as keys and the corresponding RDF files as values. These files are then stored in a local directory.

The main advantage of this method compared to using a list of stop-categories, similar to a list of stop-words, is that the categories are structured so it is only

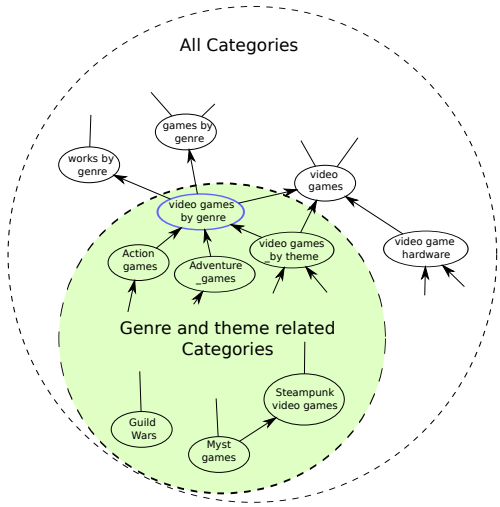


Figure 3.7: The relevant categories are all in the sub-graph from `video games by genre` even though it could be argued that themes are not genres. This is a consequence of the informal semantics in the relations.

```
function recursiveDepthFirst(node (string), closed (
    string=>dom){
  let url := "http://dbpedia.org/data/"+node
  let dom := xmlAsDom(get(url))
  closed[node] := dom
  let children := children(node)
  foreach (child in children){
    if (closed[child] != null) then
      closed := recursiveDepthFirst(child,closed)
  }
  return closed
}
```

Figure 3.8: Recursive depth first traversal. The `closed` set is implemented as an initially empty dictionary which is recursively expanded with the nodes in the sub-graphs of `node`. The function do not recurse on nodes already in `closed`. The function `get(url)` returns the web page in the argument as a string. The function `children(node)` returns the nodes of the outgoing edges of `node`.

necessary to specify a few categories as starting nodes. The remaining categories are extracted automatically. The disadvantages of this method are that *(i)* the start nodes must be chosen carefully by manually inspecting the hierarchy, and *(ii)* the informal semantics can result in undesired categories. Disadvantage *(ii)* can be avoided by choosing smaller and specific sub-graphs which are typically more consistent than big and very general sub-graphs.

The category vectors for the Greenlight submissions are constructed automatically from the entity disambiguation and they are expected to have many inaccurate categories due to incorrectly disambiguated entities. They are also generally much longer than the Steam category vectors since an entity typically has multiple associated categories. Rather than intersecting the Greenlight category vectors with the genre and theme categories as for the Steam category vectors, they are intersected with the set of categories that occurs in all Steam category vectors. The idea is that the Steam category vectors are used as model for the Greenlight category vectors so that categories which do not occur in any Steam category vector is completely omitted.

The advantage of this method is that the remaining Greenlight category vectors correlate with at least one Steam category vector. The disadvantage is that Greenlight categories are sometimes more general than Steam categories and such otherwise useful categories would be removed with this method. For example the category `Steampunk` do not match the category `Steampunk video games` and would be removed. Additionally category vectors can end up as zero-vectors which cannot be compared with anything at all.

The category intersection itself is straight-forward to implement as the function `intersection(cs,dom)` where `cs` is a set of categories and `dom` is a `DOM` with category vectors. The output is `dom` where the category vectors only contain the categories in `cs`. The function is wrapped into two modules.

```
File Intersection Module(dom,dir)
XML Intersection Module(dom1,dom2)
```

`File Intersection Module(dom,dir)` constructs `cs` from the RDF-file names in given directory `dir` and `XML Intersection Module(dom1,dom2)` constructs `cs` from the set of all categories in the category vectors of `dom2`. Figure 3.9 shows some examples of input and output of both modules.

3.3.2 Generalizing Categories

Like in [DNMO+12], the `skos:broader` taxonomy is used to *generalize* the categories in the profiles and discover implicit relations between films. The

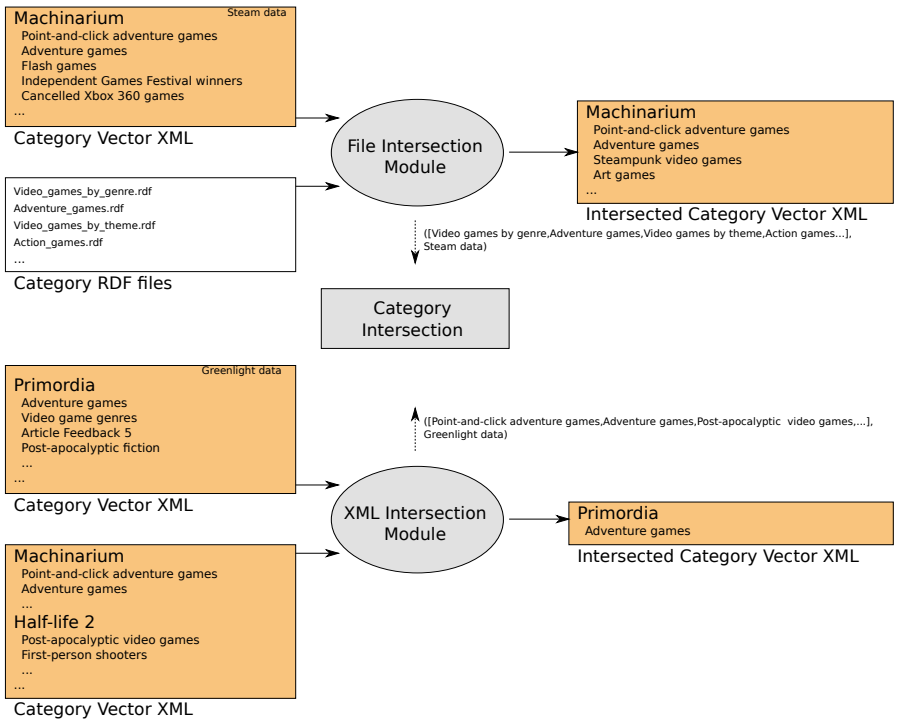


Figure 3.9: Examples of input and output of the `File Intersection Module(dom,dir)` and the `XML Intersection Module(dom1,dom2)`.

`skos:broader` is considered as *one-step transitive* such that for example given the triples

```
Machinarium dcterms:subject Steampunk_video_games.
Steampunk_video_games skos:broader Steampunk_games.
```

the following can be inferred by one-step transitivity

```
Machinarium dcterms:subject Steampunk_games.
```

This can be generalized to *n-step transitivity* and in [DNMO+12] they found that the best precision was for $n = 1$.

Intuitively highly specialized categories, such as the brand name categories `Myst` and `Guild Wars` often found in the Steam category vectors, benefits from being generalized to more broader and more common concepts with a higher correlation. On the other hand the method increases the total number of considered categories as shown in figure 3.11 which also increases the sparsity of the vectors.

In this project, category generalization is implemented as the module

`Broader Module(dom)`

which applies one-step transitivity to each category in the category vectors of *dom* by querying the DBpedia SPARQL endpoint. Suppose the given triple is g `dcterms:subject` c_0, then the set of one-step transitive categories of c_0 are defined by variable C in the SPARQL query

`select` C `where {<`c_0`> skos:broader` C`}`

For each $c \in C$, the following triples are then added to the profiles

g `dcterms:subject` c

To get n-step transitivity, the above is recursively applied to the added subjects as well. The above method can be viewed as a depth-limited graph traversal in the `skos:broader` hierarchy graph. This traversal is straight-forward to implement as the recursive algorithm `depthLimitedRec(node,closed,n)` shown in figure 3.10.

The `Broader Module(dom)` queries `depthLimitedRec(c0,1,broader(uri))` for each category `c0` in the category vectors of `dom`. The successor function `broader(uri)` returns the result of `dbpediaSparql(query)` where `query` is the SPARQL query

```
select ?C where {<uri> skos:broader ?C}
```

```
function depthLimitedRec(node (entity), closed (entity
    =>dom), n (num), children (entity=>entity set)){
  closed := union(closed,{uri})
  if (n=0) then
    return closed
  foreach concept in children(uri)
    closed := union(closed,depthLimitedRec(concept,
        closed,n-1))
  return closed
}
```

Figure 3.10: The nodes from the recursive calls on the sub-graphs are added to `closed` and the algorithm returns `closed` after recursing on all children. The `children` argument is the function which is used to generate the immediate successors.

The output categories of `depthLimitedRec(c0,1,broader(uri))` are then added to the category vector of `c0` as shown in figure 3.11.

3.3.3 Narrowing Categories

Where `Broader Module(dom)` generalizes the input category vectors to more general categories, `Narrower Module(dom)` is a module for specializing the category vectors to more specific categories.

In the `skos:broader` hierarchy it is common that a category for a concept has sub-categories for the same concept but in different contexts. For example the category `Steampunk` for the concept of "Steampunk" has the sub-categories `Steampunk games`, `Steampunk music`, `Steampunk literature` and others. The Steam raw data often contain the category dedicated to the context of video games, which in this case is `Steampunk video games` which is a sub-category of `Steampunk games`. However the Greenlight raw data often contain general categories such as `Steampunk` as the LOD entity they are derived from is not listed in very specific categories. The idea behind the `Narrower Module` is to replace these general categories with more specific categories that matches those of video games. Another way of addressing this problem would be to increase the degree of transitivity in the `Broader Module` but as noted in [DNMO+12] this will introduce noise in the Steam profiles. Following the theory that the manually assigned categories in the Steam profiles and can be used as a model for the automatically generated categories in the Greenlight profiles, the Steam

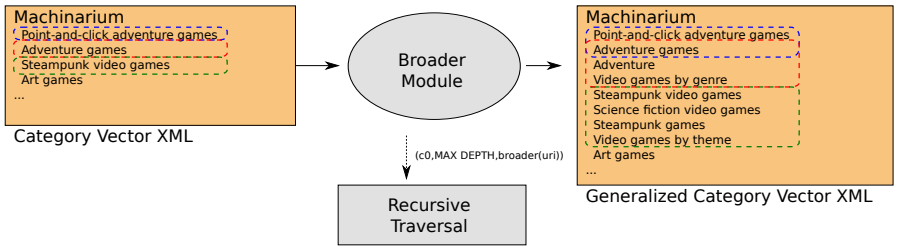


Figure 3.11: `Broader Module(dom)` adds generalized categories to the category vectors in the input. For example for `Steampunk video games` the 1-step transitive generalized categories `Science fiction video games`, `Steampunk games` and `video games by theme` are added to the vector for "Machinarium".

profiles should contain too much noise.

While it is simple to generalize categories by traversing the `skos:broader` hierarchy, it is not trivial to specialize categories as it requires additional knowledge about the context. In addition the `skos:broader` hierarchy is based on an informal thematic relation which means that even with a known context, formal logical reasoning may produce unwanted results. A very simple approximation would be to ignore the context and simply traverse the `skos:broader` hierarchy similarly to `Broader Module(dom)` and then let other modules try to filter out incorrect categories. By this method, `Narrower Module(dom)` queries `depthLimitedRec(c0,1,narrower(uri))` where `narrower(uri)` returns the results of the SPARQL query

```
select ?C where {?C skos:broader <uri>}
```

to get the 1-step transitive specialized categories of `c0`. These categories are then added to the category vector of `c0` as shown in figure 3.12.

3.4 Similarity Measure and Recommender System

The output of the pipeline of optimization modules are used without preprocessing in the similarity measure and recommender system, which lessens the computation time. The methods for the similarity measure itself and the way it is applied in the recommender system is as in [DNMO+12]. This section reintroduces the methods and presents a few additional choices in the implementation.

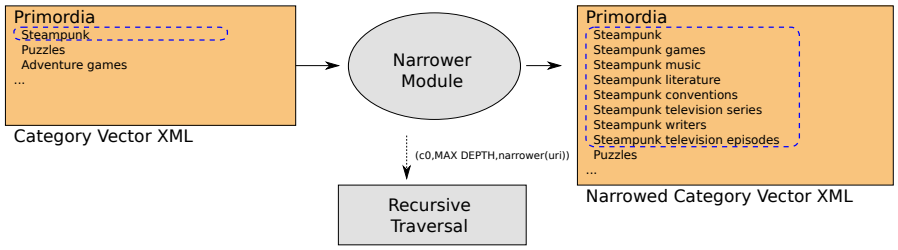


Figure 3.12: `Narrower Module(dom)` adds specialized categories to the category vectors in the input. In this example the 1-step transitive specialized categories of `Steampunk`: `Steampunk games`, `Steampunk music`, `Steampunk literature` and so on are added to the vector for "Primordia".

The output of the pipeline are two XML-files that looks like below

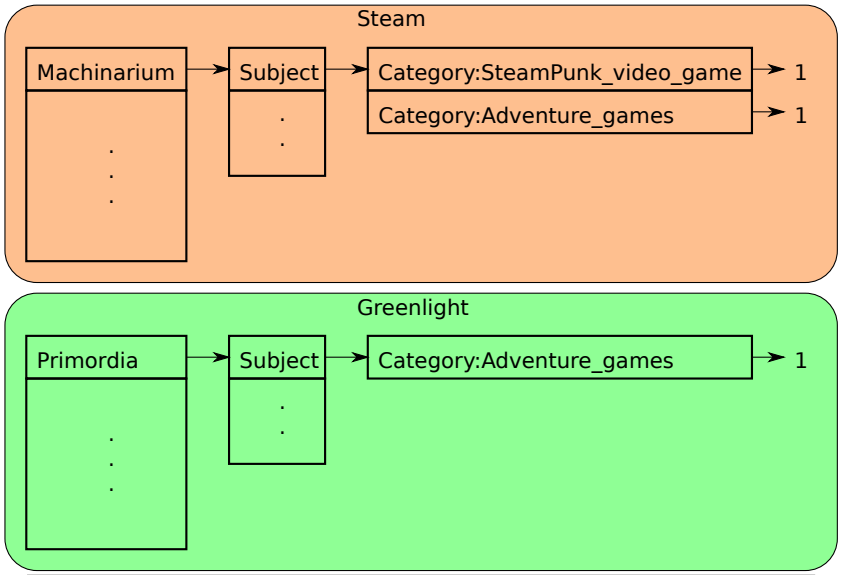
Listing 3.1: steamSubjects.xml

```
<steam>
  <game>
    <title> Machinarium </title>
    <uri> http://steampowered... </uri>
    <description> About this game: ... </description>
    <dbpediaURI> http://dbpedia.org/resource/Machinarium
        </dbpediaURI>
    <subject> Category:Steampunk_video_game </subject>
    <subject> Category:Adventure_games </subject>
  </game>
  ...
</steam>
```

Listing 3.2: greenlightSubjects.xml

```
<greenlight>
  <game>
    <title> Primordia </title>
    <uri> http://steamcommunity/greenlight/... </uri>
    <description> About this game: ... </description>
    <dbpediaURI> http://dbpedia.org/resource/Adventure_game
        </dbpediaURI>
    <dbpediaURI> ... </dbpediaURI>
    ...
    <subject> Category:Adventure_games </subject>
  </game>
  ...
</greenlight>
```

Each `game` element is converted to a profile vector by using dictionaries as shown in figure 3.13.



dcterms:subject	Adventure games	Steampunk video games
Machinarium	1	1
Primordia	1	0

Figure 3.13: The example data from above XML file snippet converted to a profile by using dynamic hash-tables. Only the `title` and `subject` tags are used to construct these tables. Since the approach only considers the `dcterms:subject` relation, the predicate dictionary always contains a single key, but the data structure allows multiple predicates to be used if available. The table below the figure shows the data interpreted as presented in [DNMO+12]. The dictionary representation avoids the problem with sparsity as 0-elements are omitted.

In the similarity measure, $sim_p(i_1, i_2)$ denotes the similarity between the LOD individuals i_1 and i_2 according to property p. If $\overrightarrow{i_1, p}$ and $\overrightarrow{i_2, p}$ are the vectors of triple objects for i_1 and i_2 with respect to property p, then

$$sim_p(i_1, i_2) = \frac{\overrightarrow{i_1, p} \cdot \overrightarrow{i_2, p}}{|\overrightarrow{i_1, p}| \cdot |\overrightarrow{i_2, p}|}$$

With the dictionaries of figure 3.13, the vectors are found by using combined dictionary operations

$$\overrightarrow{i_1, p} := lookUp(p, lookUp(i_1, d_1)) = \begin{pmatrix} k_1 \Rightarrow v_1 \\ \vdots \end{pmatrix}$$

$$\overrightarrow{i_2, p} := lookUp(p, lookUp(i_2, d_2)) = \begin{pmatrix} l_1 \Rightarrow w_1 \\ \vdots \end{pmatrix}$$

where d_i is one of dictionaries in figure 3.13. The dot product is computed by iterating through k_i and summing up $v_i \cdot w_i$ when $k_i = l_i$.

In the approach of [DNMO+12], a user profile u is a set of tuples (i, v) where i is an LOD individual and v is either 1 or -1, depending on whether the user *likes* or *dislikes* i. The recommender system then computes an overall relevance $\overrightarrow{r(u, i_1)}$ of individual i_1 for the user profile u by averaging the similarity measure

$$r(u, i_1) = \frac{\sum_{(i_2, v) \in u} v \cdot \frac{\sum_{p \in P} \alpha_{p \in P} \cdot sim_p(i_1, i_2)}{|P|}}{|u|}$$

where P is the set of properties and $alpha_p$ is a weight for the individual property p. In this project, the user profile u' is a set of LOD individuals for the games owned by the user and thus there are no i that the user *dislikes*. In [DNMO+12] they consider approaches for choosing values for α_p but since there is only one property in this project, it is set to the constant 1. Taking the above into consideration, the implemented formula $\overrightarrow{r'(u, i_1)}$ is simplified to

$$r'(u', i_1) = \frac{\sum_{i_2 \in u'} \frac{\sum_{p \in P} sim_p(i_1, i_2)}{|P|}}{|u'|}$$

The recommender system uses the function `RECOMMEND` which takes as input *(i)* a user profile *(ii)* the Greenlight titles to recommend from, *(iii)* a Steam game dictionary, and *(iv)* a Greenlight submission dictionary. It then sorts the Greenlight titles by r' by using a binary max-heap.

```
input: u' (string set), titles (string set),
     greenlight_D ((string,(string,(string,integer)))),
     steam_D ((string,(string,(string,integer))))
output: ranking (string heap)
RECOMMEND:
let ranking := empty max-heap
foreach t in titles{
    let v := r'(u',t,greenlight_D,steam_D)
    ranking := insert((t,v),ranking)
}
return ranking
```

Chapter 4

Evaluation

This chapter presents the evaluation of the work presented in chapter 2 and 3 and discusses the results and further development. Section 4.1 presents the evaluation data sets and the metric. Section 4.2 presents the evaluation results and section 4.3 presents the discussion of the project.

4.1 Evaluation Strategy

The evaluation requires two groups of data sets. The first group consists of goal data which the output is evaluated against. These data sets are considered in the evaluation metric. The second group consists of input data sets with product profiles. These data sets are necessary to produce an output given a user profile. This section presents how both groups of data sets are extracted from the Web and the evaluation metric.

4.1.1 Goal Sets

As outlined in the introduction, both the similarity measures and the recommender system is evaluated by the *average precision* of the rankings they output.

When measuring the average precision of a ranked list, each item in a ranked list is classified as either *relevant* or *not relevant* and the average precision is high when the relevant items are among the top of the ranked list. Hence the relevant items must be defined for both the similarity measure and the recommender system.

4.1.1.1 Similarity Measure Goal Set

For the similarity measure, the evaluated output is a similarity ranking of Steam games for a given Greenlight submission. In this case, the relevant items are the Steam games which are similar to the Greenlight submission. In this project, such information is extracted from the video game encyclopedia at the website http://giantbomb.com. This encyclopedia is maintained in a similar way to Wikipedia but is focused on video games. The available data for each game in the website are as follows

1. A text description
2. A wikipedia-like "infobox" with semi-structured data
3. Pictures/video
4. "Related pages" with additional data.

Among the "related pages" is a page of similar games where the website users can enter games that they believe to be similar. The idea is to use the data in these pages as a human made similarity classification which defines candidates for the relevant items of each Greenlight submission. Among these candidates only those that satisfy the following requirements becomes relevant items

1. The game must be available in Steam
2. The game must have a corresponding entity in DBpedia

The data set is extracted manually by choosing a set of Greenlight submissions which occur in the encyclopedia and which have similar games satisfying the above requirements. The result is a table as shown in table 4.1 which is stored in an XML file. Any similar Steam games which were incorrectly mapped to Dbpedia entities were corrected manually. Although the goal set is rather small, no other similar data set for this domain was found at the time of writing.

Submission title	Similar games
Primordia	[Machinarium, Gemini Rue, ...]

Table 4.1: Example of the extracted similarity measure goal set. For each submission title there is a list of similar games.

User	Profile	Relevant items
user 1	[Ace of Spades, Alice: Madness Returns,...]	[Primordia, Postal 2]
user 2	[Machinarium,Half-life 2,...]	[Primordia]

Table 4.2: Example of the extracted recommender goal set. For each user are their Steam games (profile) and the Greenlight submissions that they like (relevant items).

4.1.1.2 Recommender System Goal Set

The recommender system in [DNMO+12] was evaluated using *MovieLens*, a historical dataset of user ratings for film recommender systems. A similar fitting dataset was not found at the time of writing so instead the goal set is created semi-automatically from data on the Web. Although the ratings the users can give in Greenlight are not public data, any given user in Steam can choose to make their list of purchased games public at `http://steamcommunity.com/id/ID/games` where *ID* is a unique ID for the user. For this evaluation, a purchase is interpreted as a positive rating so that the user ratings can be extracted from these lists. The users who bought games from the considered Greenlight submissions are extracted from the official user group of each game at at `http://steamcommunity.com/games/APP`, where `APP` is a unique ID for the game. The members of this user group are then available at the web page `http://steamcommunity.com/games/APP/members` and they can be extracted by inspecting the web page with a PHP script. The only extracted users are those who bought one of the considered Greenlight submissions. In total, the result is a table as shown in table 4.2 which is stored as an XML file.

4.1.2 Input Data Sets

The input data are two sets of products, the released products which users may have rated and the unreleased products that are unknown to the users. Both of these sets are extracted from the Web and processed so that they can be applied with the methods presented in chapter 2 and 3.

4.1.2.1 Steam Games

In this project the released products are Steam games. The data set is constructed in the following way.

1. Steam game titles and descriptions are extracted from the Web.
2. Each title is mapped to the corresponding Dbpedia entity ex.

   `Machinarium` ↦ `http://dbpedia.org/resource/Machinarium`

For step 1, the titles and descriptions are extracted by crawling the Web-based store of Steam games. Every game in Steam has an HTML web page in `http://store.steampowered.com` which contains the title and a description of the game. Links to these web pages are found by using the "search" feature of the Steam store for all video games and then parsing the resulting HTML web pages for the links of each title. Only the games available for Windows are considered as they make up the majority of the titles. This method only uses the freely available data from Steam although some games are skipped as their web page require an additional age check. The data set is stored in an XML file as below

Listing 4.1: steamDescriptions.xml

```
<?xml version="1.0" encoding="utf-8"?>
<steam>
  <game>
    <uri>http://store.steampowered.com/app/40700/?snr=1
        _7_7_230_150_36</uri>
    <title> Machinarium <title>
    <description> About this game: ... </description>
  </game>
  ...
</steam>
```

For step 2, the method used in [DNMO+12] is applied. Every DBpedia individual has a `rdfs:label` property which defines a human-readable name of the entity in different languages. These labels are compared with the titles by their Levensthein distance to find the most fitting DBpedia entity. The entity-label pairs of video games are extracted from the DBpedia SPARQL endpoint. Since Steam mostly contain games for the Windows platform a large amount of entities can be skipped by including a restriction of the `computingPlatform` property to `Microsoft Windows`.

```
select ?uri ?label where {
```

uri	label
http://dbpedia.org/resource/StarCraft	StarCraft
http://dbpedia.org/resource/Trine_(video_game)	Trine (video game)
http://dbpedia.org/resource/Machinarium	Machinarium

Table 4.3: Example of extracted entity-label pairs from DBpedia.

```
<?xml version="1.0" encoding="utf-8"?>
<steam>
  <game>
    <uri>http://store.steampowered.com/app/40700/?snr=1
        _7_7_230_150_36</uri>
    <title>Machinarium</title>
    <description>About the ... </description>
    <dbpediaURI>http://dbpedia.org/resource/Machinarium</dbpediaURI
        >
  </game>
  ...
</steam>
```

Figure 4.1: steamDbpedia.xml

```
  ?uri a dbpedia-owl:VideoGame.
  ?uri rdfs:label ?label.
  ?uri dbpedia-owl:computingPlatform dbpedia:Microsoft_Windows.
  FILTER langMatches(lang(?game), "EN").
}
```

The result of this SPARQL query from DBpedia is an XML encoded table as shown in table 4.3. For this evaluation, the best matching pair for a given game title is found by a simple all-pair comparison. Since the label often has the format "*title* (video game)", the algorithm measures the Levensthein distance both with and without " (video game)" appended to the game title. Unfortunately many of the extracted game titles contain non-ASCII symbols or have an appended text such as ": Steam Edition" which lowers the precision of the method significantly. To increase the precision, the games where the best match exceeds a given *limit* are not mapped to an entity. Afterwards the games that occur in the goal data are corrected manually and the games that are not mapped to an entity are completely removed from the data set. The final data set is stored in an XML-file as shown in figure 4.1. Note that the data set contains all data necessary for both the method of chapter 2 and the method of chapter 3. This ensures that both methods are evaluated with the same input data.

4.1.2.2 Greenlight Submissions

The second set of input data are the unreleased products which do not have corresponding LOD entities. In this project these products are the Greenlight submissions and as these are not part of the Steam store they are not extracted with above methods. The data set is extracted by manually finding a set of submissions which can be evaluated with the goal data. For the evaluation, the recommender system only recommends among the Greenlight submissions released as Steam games as the goal set covers these games. These games have a web page in the Steam store but since the recommender system is target towards unreleased games, the evaluation considers only the textual description from the Greenlight submission.

In this evaluation the submissions of 11 released Greenlight games were extracted by parsing their HTML web-pages with PHP. The resulting XML-file was then used in the profiling as presented in chapter 3.

Listing 4.2: greenlightDescriptions.xml

```
<?xml version="1.0" encoding="utf-8"?>
<greenLight>
  <game>
    <title>primordia</title>
    <uri>http://steamcommunity.com/sharedfiles/filedetails/?id
        =108108057</uri>
    <description>An adventure in the tradition of ...</description>
  </game>
  ...
</greenLight>
```

4.2 Results

Below are tables with the results of the evaluation and in-depth explanations of the evaluation methods. Table 4.4 and 4.5 show an overview of the evaluation data constructed with the methods from section 4.1. The full evaluation datasets are in the appendix.

Table 4.6 shows the average precision of each of the evaluated similarity measures and table 4.7 shows the average precision of the recommender system with both LOD-based similarity measures.

Data set	Quantity
Steam games	644 games
Greenlight submissions	12 submissions

Table 4.4: Input data sets.

Submission	# Similar games	# Users
Primordia	6	9
Postal 2	0	29
Waking Mars	1	27
Miner Wars 2081	0	23
Towns	5	35
Fly'n	3	28
AIE	2	9
Miasmata	9	22
Giana Sisters	2	28
Air Buccaneers	4	29
Forge	6	35
Mcpixel	0	27
Total	38	287*

Table 4.5: Goal data sets. *Notice that some users bought more than one of the evaluated Greenlight submissions.

Title	TD-Matrix	LSA	LOD 1	LOD 2
Primordia	0.032	0.020	0.089	0.032
Postal 2	N/A	N/A	N/A	N/A
Waking Mars	0.001	0.005	0.002	0.003
Miner Wars 2081	N/A	N/A	N/A	N/A
Towns	0.015	0.012	0.034	0.006
Fly'n	0.007	0.018	0.048	0.032
AIE	0.021	0.009	0.073	0.034
Miasmata	0.095	0.018	0.023	0.053
Giana Sisters	0.034	0.006	0.103	0.026
Air Buccaneers	0.073	0.008	0.012	0.006
Forge	0.048	0.012	0.007	0.01
Mcpixel	N/A	N/A	N/A	N/A
Overall Average precision	**0.036**	**0.012**	**0.043**	**0.022**

Table 4.6: Evaluation results for the similarity Measure.

Title	LOD1	LOD2
Primordia	0.3	0.29
Postal 2	0.1	0.09
Waking Mars	0.36	0.35
Miner Wars 2081	0.12	0.1
Towns	0.11	0.12
Fly'n	0.9	0.87
AIE	0.22	0.18
Miasmata	0.23	0.21
Giana Sisters	0.28	0.23
Air Buccaneers	0.15	0.15
Forge	0.44	0.69
Mcpixel	0.17	0.13
Overall Average Precision (by user)	**0.28**	**0.30**

Table 4.7: Evaluation results for the recommender system.

4.3 Discussion

In this section the results from section 4.2 are evaluated and possible improvements to the methods are discussed.

4.3.1 Dataset Evaluation

Since the datasets used for the evaluation were created for this project it is necessary to discuss their reliability. The similarity measure goal set is constructed semi-automatically from the similarity classifications made by the users of the Web page http://giantbomb.com. Given that the Web site is maintained by the whole community of users the accuracy should be similar to that of a site like http://wikipedia.com. The extracted dataset suffers from sparseness however as seen in table 4.5 where some submissions only have a few similar games. One way to reduce the sparseness would be to automatically extend the dataset with a transitivity rule for similarity and a heuristic based on some of the other features in the dataset. For example, given that $sim(A, B)$ and $sim(B, C)$, transitivity would generate a candidate $sim(A, C)$ which would then be validated by a heuristic method.

The recommender system goal set is also constructed semi-automatically by extracting data from the Web however it has some issues as well. Despite evaluating 644 user profiles, many of the users only purchased one or two of

the 9 evaluated Greenlight submissions. Combined the low number of evaluated Greenlight submissions the recommender goal set is rather sparse. Secondly the data relies on the assumption that a purchase equals a positive user rating which has not been validated. A different approach to construct a goal set would be to generate a number of user profiles and then identify the relevant items for each of the generated user profiles.

4.3.2 Result Evaluation

Despite the discussed weaknesses of the evaluation datasets, the results do show some trends to be observed. With respect to the similarity measure evaluation, the LOD approach with short post-processing performs overall slightly better than the simple baseline approach with term-document vector correlation. Surprisingly the LSA based similarity measure performs worse than the simple term-document vector correlation. Potential causes may be

Bad query vectors The words used to form query vectors for the Greenlight submissions are computed separately from the Steam game term-document matrix. This is different from the LSA example in the introduction where the words were the document-vector of the document.

Suboptimal Lingpipe settings Besides the number of desired singular values, the Lingpipe library for performing SVD requires several settings to be set. For this evaluation the number of singular values were 2 and the remaining settings were found by empirical testing.

The results also show that expanding the category vectors with the category hierarchy do not improve the performance. One of the issues in the expansion method is that all categories are expanded equally which assumes a certain equality in the generality among the categories. This is typically incorrect as categories high in the hierarchy are intended to more general than those lower in the hierarchy. Intuitively, it is often very specific terms that should be generalized so that they can be compared to other terms. The challenge is then to estimate the generality of a category which is not trivial when the hierarchy is not balanced.

With respect to the recommender system evaluation, the short post-processing performs overall slightly worse than the full post-processing, despite performing worse as a similarity measure. In figure 4.2 the results of table 4.6 and 4.7 are combined to show the relation between similarity performance and recommender system performance. In several cases the short post-processing gives better

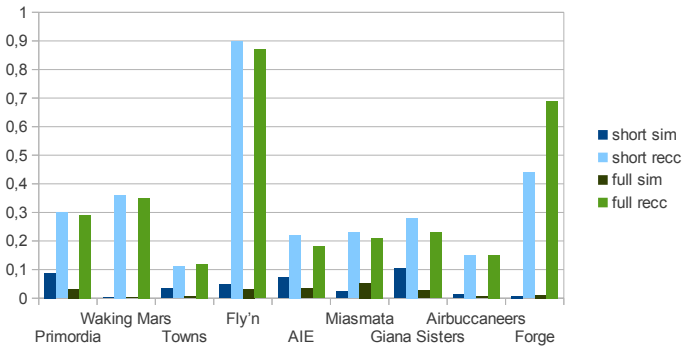


Figure 4.2: Evaluation results visualized as a bar chart to compare similarity measure performance against recommender system performance.

similarity measure results than the full post-processing, but the corresponding recommender system results are barely better and in some cases even worse. Such a development could signal that improving the similarity measure would not improve the recommender system performance drastically, however given the issues in the evaluation data presented above this conclusion may not be accurate.

4.3.3 Further Method Improvements

As outlined in [RRS11] the qualities of a recommender system goes beyond the the quality of the recommendations. In particular the interaction between the recommender system and the user is a major issue which have only lightly been addressed with the shown web based demo. However this section only discusses possible improvements to the quality of the recommendations as it has been the major focus of this project. The improvements are addressed accordingly to the issues of content-based recommendations introduced in chapter 1.

4.3.3.1 Multiple Properties

In [DNMO+12] they measured an overall improved performance when considering additional properties beside the categories and that may apply to the recommender system in this domain as well. In detail, the feature could address the following issues of content-based recommendation

Limited Content Analysis The measured similarity would be an average of the similarity across all properties and that would reduce any noise from the category vectors.

Overspecialization It would be possible to make a formalized notion of product diversity by using property weights. For example the recommender system may recommend a product which only matches the user profile on a few important properties.

The problem with this feature is that it requires disambiguation of properties for the unreleased products which is yet a difficult challenge.

4.3.3.2 Cross-domain Recommendation

A content-based recommender system with cross-domain recommendation would be to measure similarity between products of different domains. For example if the user is known to like books or films about a given topic, the recommender system may recommend video games about that topic. In [HKP+10] the authors identifies potential gains and issues of cross-domain recommendation. Cross-domain recommendation could address the following issues of content-based recommendation.

Overspecialization The recommender system can get a larger profile of the user and may find correlation with new properties from different domains.

New user problem As the recommender system can reuse user ratings from other domains, the user no longer have to provide user ratings for each domain.

Chapter 5

Conclusion

In this project a content-based recommender system for the given domain has been evaluated with datasets extracted from the Web with semi-automatic methods. The datasets were found to be rather sparse but methods have been presented to improve upon this issue. The recommender system and the extracted datasets have been made available on a web-page.

The problem of measuring similarity between products in the domain have been evaluated with two groups of methods. In the first group the problem was solved by traditional natural language processing methods and products were only represented as documents. In the second group the problem was solved by combining natural language processing methods with the data available in DBpedia. To my knowledge there has not been an attempt at combining natural language processing and the Linked Open Data in this way before. The method that performed overall best as a similarity measure belonged to the second group.

The problem of recommending products in the domain have been evaluated only with the methods in the second group. The results suggest that improving the similarity measure may not improve the recommender system performance significantly. A promising extension would be to consider cross-domain recommendation, where the recommended products are compared with products of different type from different sources on the Web.

Appendix A

XML Tags

This appendix is an overview of the XML tags used in the input datasets and demonstration of the usage.

Tag	Meaning
`<steam>`	root for XML files about Steam games
`<greenlight>`	root for XML files about Greenlight submissions
`<game>`	an individual Steam game or Greenlight submission
`<title>`	title of the parent `<game>`
`<uri>`	uri of the parent `<game>` in Steam or Greenlight
`<description>`	textual about the parent `<game>` from Steam or Greenlight
`<dbpediaURI>`	uri of a Dbpedia entity related to the parent `<game>`
`<subject>`	uri of a Dbpedia category related to the parent `<game>`

```
<steam>
  <game>
    <title> Machinarium </title>
    <uri> http://steampowered... </uri>
    <description> About this game: ... </description>
    <dbpediaURI> http://dbpedia.org/resource/Machinarium
        </dbpediaURI>
    <subject> Category:Steampunk_video_game </subject>
    <subject> Category:Adventure_games </subject>
  </game>
  ...
</steam>
```

Bibliography

[AT05] Gediminas Adomavicius and Alexander Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Trans. on Knowl. and Data Eng.*, 17(6):734–749, June 2005.

[BHBL09] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

[BLK+09] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. Dbpedia - a crystallization point for the web of data. *Web Semant.*, 7(3):154–165, September 2009.

[Bur07] Robin Burke. The adaptive web. chapter Hybrid web recommender systems, pages 377–408. Springer-Verlag, Berlin, Heidelberg, 2007.

[DDF+90] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*, 41(6):391–407, 1990.

[DNMO+12] Tommaso Di Noia, Roberto Mirizzi, Vito Claudio Ostuni, Davide Romito, and Markus Zanker. Linked open data to support content-based recommender systems. In *Proceedings of the 8th International Conference on Semantic Systems*, I-SEMANTICS '12, pages 1–8, New York, NY, USA, 2012. ACM.

[HKP+10] Benjamin Heitmann, James G. Kim, Alexandre Passant, Conor Hayes, and Hong-Gee Kim. An architecture for privacy-enabled user profile portability on the web of data. In *Proceedings of the 1st International Workshop on Information Heterogeneity and Fusion in Recommender Systems*, HetRec '10, pages 16–23, New York, NY, USA, 2010. ACM.

[HLA12] Sebastian Hellmann, Jens Lehmann, and Sören Auer. Towards an ontology for representing strings. In *Proceedings of the EKAW 2012*, Lecture Notes in Artificial Intelligence (LNAI). Springer, 2012.

[MJGSB11] Pablo N. Mendes, Max Jakob, Andrés García-Silva, and Christian Bizer. Dbpedia spotlight: shedding light on the web of documents. In *Proceedings of the 7th International Conference on Semantic Systems*, I-Semantics '11, pages 1–8, New York, NY, USA, 2011. ACM.

[NA11] Axel-Cyrille Ngonga Ngomo and Sören Auer. Limes: a time-efficient approach for large-scale link discovery on the web of data. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume Three*, IJCAI'11, pages 2312–2317. AAAI Press, 2011.

[RRS11] Francesco Ricci, Lior Rokach, and Bracha Shapira. Introduction to recommender systems handbook. In Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor, editors, *Recommender Systems Handbook*, pages 1–35. Springer, 2011.

[SKW07] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 697–706, New York, NY, USA, 2007. ACM.