Universitat de Lleida
Escola Politècnica Superior
Màster en Enginyeria del Informàtica

# Outfit recommendation system

Autor/a: Robert Sanfeliu Prat
Director/s: Carlos Ansótegui Gil

Setembre 2013

University of Lleida

Master in Computer Engineering

# Outfit recommendation system

*Author:*
Robert Sanfeliu Prat

*Supervisor:*
Carlos Ansótegui Gil

September 2013

# *Acknowledgements*

This project has been made under the sincere guidance of Carlos Ansótegui. I would like to thanks my teachers, parents, my girlfriend and my dear friends for helping me in this project.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

This chapter gives a brief introduction to the topic discussed in this project. First, the reasons that lead to the execution of this project are described. Then, the topic of this research is introduced. After that, the set of objectives to accomplish are defined. This chapter ends showing how this document is organized.

## 1.1 Motivation

Nowadays, the way we dress is an important aspect of ourselves that may influence the opinion that the society has of us. Fashion is an ever-changing world where trends appear and disappear each season, and what is ”in” today may be out of date tomorrow. Also, due to the reduction of the cost of garments, people tend to own a large quantity of garments. Those three facts, make it really hard for some people decide what to wear or how to style a certain garment. That problem is so prevalent that many fashion magazines have a specific section dedicated to answer style questions from the readers. Also some on-line fashion communities have been build with the sole propose of asking and giving advice on what to wear, or have this feature as an important one. Fashism[9], ”Go try it on”[14] are two examples of fashion communities build for their members to ask and give advice on how to dress. Polyvore[21] and others offer this as an important feature.

Whilst giving an answer to the user needs, all of the presented solutions have a major drawback: they are not immediate. The time between when the user asks the question and when he receives an answer ranges from weeks (fashion magazines) to hours(on-line fashion communities). That makes the existing advice sources less useful as, usually, the user, who is in the process of selecting an outfit, needs immediate feedback to make the decision.

Our motivation is to develop a system capable of automatically making recommendations on how to style a certain garment.

## 1.2 The topic of this research

In this project we focus in the application of recommendation systems technology on the fashion industry. The aim is to develop a recommendation system that given a garment, automatically generates an outfit with that particular garment. The generated outfit should potentially match the user taste. The problem we are trying to solve is present in our everyday life and could be informally stated as ”How do I style that garment?”.

Recommender systems are agents that produce items recommendations (books, films, songs, etc.) for an user. Their recommendation is based on a prediction of how interesting that item will be for the user. That prediction can be made using three approaches: using a model of the characteristics of the items, using a model of the user social environment or a mix of both (Schafer et al. [23]).

Recommender systems have become an important research area in the later years. Part of this interest on the topic is due to the large quantity of potential applications that the research has.

One of this areas of application is the recommendation in e-comerce environments. In such applications, the goal is to present the user items from the on-line store that might be relevant to him and, thus, a potential purchase. This recommendation systems are a keystone for many e-comerce sites such as Amazon[2] or Netflix[18]. As an example of how important the recommendation systems are for e-comerce sites, Netfilx undertook a 1M$ prize contest[19] for the best recommendation system that over-performed the one that the company had.

This project aims to develop a recommendation system that takes profit from the data generated by an on-line fashion community. When queried with a garment, the system will use that data to create an outfit that appeals to the user taste.

## 1.3 Objectives

In this section, the main objectives for the project are presented and described.

The main objectives of this projects are:

- Develop a data gathering agent

- Automatize the evaluation of items similarity
- Apply learning algorithms to detect similar items
- Develop a recommendation system

The proposed recommendation system relies on the data captured from a social network where its members combine garments from a virtual wardrobe to create outfits. Those outfits are exposed to the other members judgement, who can express their approval by clicking a ”I like it” button. The community also gives the chance for members to follow other members, establishing a connection between them and thus creating a social network.

Given that our recommendation system bases its recommendations on outfits created by real people, this data needs to be captured. The first goal is to develop an autonomous agent that surfs the web and captures all the relevant data that will be used later. The captured data is: the garments in the wardrobe, the outfits created with them and information about the community members (which outfits and garments like and which other members are following).

The captured data contains a set of garments that are used to create the outfits. On average, that garments are used in two or three different outfits. That poses a problem, as when having to recommend outfits with a certain garment, that garment is going to be used in few outfits. Also, each outfit created with the query garment will provably be unique (there will not be two outfits with the same garments). This lack of repetition is even worse if only the outfits relevant to the user, those created or liked by the user or his closest community members, are taken into account. In order to overcome this difficulty, we exploit the fact that some garments share some attributes (color, shape, fabric, etc.) and are treated as similar by people.

The data of the garments coming from the fashion community only consists of a categorization, a textual description and an image. In order to apply learning algorithms to detect similar garments, some attributes values needs to be extracted from the garment data. These attributes values are extracted applying image processing techniques and natural language analysis techniques.

With the application of learning techniques the original set of garments is divided into subsets of similar ones. Those subsets are what we call abstract garments. Whenever a concrete garment appears in an outfit, that concrete garment can be replaced by its abstraction. Then the outfits with abstract garments can be grouped into similar ones, producing abstract outfits. Two outfits ($o_i$ and $o_j$) can be considered similar if they have the exact number of garments and for each abstract garment in $o_i$ there is the same garment in $o_j$. Given the abstraction of the query garment, the provability of the user (or its closest community members) to have liked an abstract outfit with that abstract garment is bigger than the provability that they have liked a concrete outfit with the concrete query garment. This provability increment gives provides, to the recommendation system, more information about the user taste, and as a consequence, the ability to produce better recommendations.

The recommendation system takes as a query a garment (picture, description and categorization) and the user information. The result of a query is an outfit that is expected to fit the user taste. To accomplish this task, the abstract outfits with the abstraction of the query garment are selected and scored. The computed score represents how appealing is a certain abstract outfit to the user, its closest community members and the whole community. Based on each abstract outfit score, one is selected and translated to a concrete outfit. This concrete outfit is finally presented to the user as an answer to its query.

## 1.4 Document structure

This document is structured in 6 chapters:

**Chapter 1, Introduction** In this chapter, First, the reasons that lead to the execution of the project are described. Then, the topic of the research is introduced. After that, the set of objectives to accomplish are defined. This chapter ends with the description of the document structure.

**Chapter 2, State of the art** In this chapter, the relevant work on recommendation systems in general and outfit recommendation systems in particular is analysed.

**Chapter 3, Polyvore** This chapter introduces the social network where the data for the recommendation system is gathered from.

**Chapter 4, Design** In this chapter, an overview of the design of the whole system is made. First a brief description of the whole recommendation system is given. Then the modules that conform it are presented, namely the data gathering, garment similarity, garment clustering and recommendation modules. After that, a more detailed description is given for each module.

**Chapter 5, Implementation** In this chapter, details of the implementation of the recommendation system are given.

**Chapter 6, Performance analysis** In this chapter, the performance of the recommendation system is assessed.

**Chapter 7, Installation and execution** In this chapter, the process for installing the recommendation system is described. After that, the most relevant tasks that can be performed with the system are described.

**Chapter 8, Project planing and costs** In this chapter, the planing followed to execute the project is described. After that, the costs of the project are listed.

**Chapter 9, Conclusions** In this chapter, the conclusions of this work are presented and some ideas for future work are outlined.

# Chapter 2

# State of the art

In this chapter, the relevant work on recommendation systems in general and outfit recommendation systems in particular is analysed.

Recommender systems are agents that produce items recommendations (books, films, songs, etc.) for an user. Their recommendation is based on a prediction of how interesting that item will be for the user. The recommendation systems can be divided into three categories (according to the approach used for assessing the user potential interest in the items): content-based, collaborative or hybrid (Schafer et al. [23]).

**Content-based recommendation systems** base the prediction of the user interest in a new item on historical data of the user interest in other items. Based on the characteristics of the items, the recommendation system is capable of finding similar items to others that the user is known to like. In other words, the system recommends items similar to other items that the user likes. This kind of recommendation systems often require the items to be characterized using a set of attributes with machine readable values (numbers or categories).

**Collaborative recommendation systems** base their prediction on the similarity between the users behaviour (bought items, seen items, liked items, etc.). Being able to find similar users to the target user (the user to whom make a recommendation), the system can recommend items that are known to be interesting for these similar users. This kind of recommendation systems are based on the idea that what is interesting for someone like us, will provably be interesting for us.

**Hibrid recommendation systems** combine both approaches (content-based and collaborative).

In what follows, some of the most relevant applications of recommendation systems technology in the world of fashion are described and analysed.

Sekozawa [24] proposes a garment recommendation system with an hybrid approach. The clustering of the garments using a k-means clustering algorithm (content-based approach). The system discovers the user taste using analytical hierarchical process (AHP) and performs market basket analysis to find tendencies on buyers with similar taste (collaborative approach).

The system requires a database of garments characterized by a qualitative value (from ”very bad” to ”very good”) for a series of attributes (criteria). The attributes used are: slim build, normal and pump build (for silhouette); simple, normal and elegant (for design); conservative, contemporary and a avant-garde (for sensitivity) and individual system, casual system, mode system (for system). Those attributes values need to be provided by an human expert. The attribute-based characterization of the garments is later used to cluster them using a k-means clustering algorithm.

The users taste is captured using an AHP process. The user is requested for a preference value for each pair of the attributes defined before. Using these preference values, the users with similar tastes can be detected.

To apply the market basket analysis, the system requires data of garments bought together by the users. The data required has the form $(garment_1, garment_2, garment_3, confidence)$ where $confidence$ is the confidence value of the purchase of $garment_3$ when the user has purchased $garment_1$ and $garment_2$. Using the market basket analysis, the system can make recommendations based on purchases made by similar users.

The major drawback of this recommendation system is the elevated amount of interaction that it requires. First of all, each garment in the system needs to be analysed by an human to give values to all of the observed attributes. Although feasible for small amounts of garments, this approach becomes less practical when the database of garments grows to several hundred of thousands of garments (the size of the database used for our recommendation system). The system also requires the user to explicitly state his taste through a series of surveys. The need for the user to explicitly input his taste is a drawback (as reported in the conclusions of the work).

Harada, Okamoto, and Shimakawa [12] propose an outfit recommendation system based on the similarity between garments (content-based approach). The authors propose to build a complex model of the outfits. The garments are characterized using the concepts of type and style. The type of a garment is characterized by a category (cardigan, long-sleeve t-shirt, etc.) and the covered area (long skirt, knee length skirt, half pants, etc.). The style of a garment is characterized by the attributes colour, pattern, material and shape. An outfit is characterized by the structure of the garments that compose it and the style of the outfit. The structure of the outfit is composed by two hierarchies: one for the upper part of the body and other for the lower. The position of each garment that composes the outfit, in any of both hierarchies, is directly related to its physical position in the outfit (the the layering). The style of an outfit is characterized using the attributes: colour, pattern, material and shape. The value of each of these attributes is derived from the values of the same attribute of the garments that compose the outfit. The amount each garment contributes to the values of the outfit attributes depends on the position that the garment occupies in the structure of the outfit(the layering).

The system relies on historical data of outfits (represented as already described) worn by the user. The system also requires what the authors call a policy graph. A policy graph is a graph that represents the valid combinations of garment types for an user depending on the position of those garments (layering). For instance, using the policy graph, an user might state that she does not want to wear a pair of leggings beneath a mini-skirt.

With all that information, the system is capable of extract the most common outfits worn by the user. Based on those common outfits, the system presents new outfits introducing slight variations.

This approach to the outfit recommendation problem proposes a complete outfit characterization. Also, the policy graph provides a deep understanding of the user taste. Both aspects combined make it possible, for the recommendation system, to create new outfits that will fit the user taste. Despite of that, the major drawback of the system is the amount of data required for each single outfits. Describing the outfits with the proposed structure is tedious for an end user and can not be easily automatized.

Shen, Lieberman, and Lam [25] propose an hybrid recommendation system focused on making recommendations to the user depending on the occasion he is attending (a dinner with the boss, a night out with friends, etc.). The garments are characterized by a brand, a type (jeans, trowsers, etc.) and a style. The style is a six-tuple of values (between 0 and 10). Each value represents the accordance of the garment with an style (luxurious, formal, funky, elegant, trendy and sporty).

To query the system, the user inputs a textual description of the occasion he is going to attend. With the application of common sense reasoning on the query text, appropriate garments are selected. This selection is done inferring the desired style from the query and searching garments with that style. The system learns with its usage, if the user decided to combine a T-shirt with a pair of trousers and sport shoes to go out with his friends, next time the user asks for an outfit for the same occasion, the system will present a similar outfit.

The major novelty of this approach is that it takes into account the specific situation and the weather. It also proposes a more user-friendly interface, given that, it allows to query the system using natural language. The major drawback of this recommendation system is that the values for the six-touple used to characterize a garment style need to be provided by an human.

In general, the major drawback presented by the analysed recommendation systems approaches is the necessity of many human interaction. This interaction is required either for providing a characterization of the outfits and garments, or a characterization of the user taste. This high level of interaction from the users make it difficult for the presented approaches to be applied into real world scenarios where there are hundreds of thousands of garments and outfits.

# Chapter 3

# Polyvore

This chapter introduces the social network where the data for the recommendation system is gathered from.

## 3.1 Polyvore

Polyvore is a social network centred on the world of fashion. Through the web page http://www.polyvore.com/ its users can upload new garments providing a picture, a small textual description an its category. All the uploaded garments conform what we call the digital wardrobe. Most of the pictures come directly from the manufacturer web site, and are pictures of the garment alone (no model) in front of a white background.

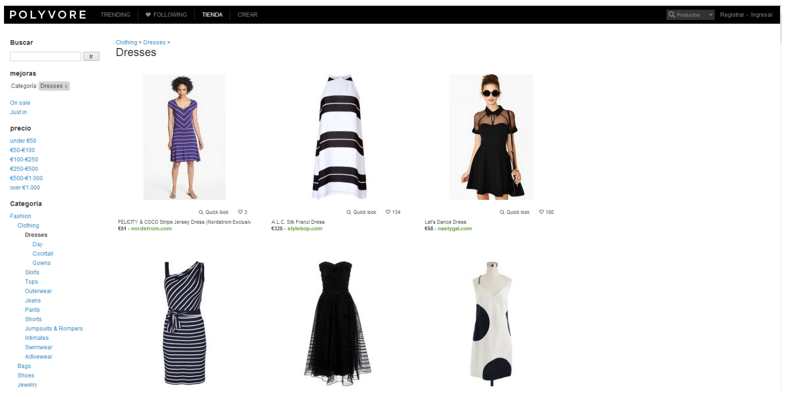
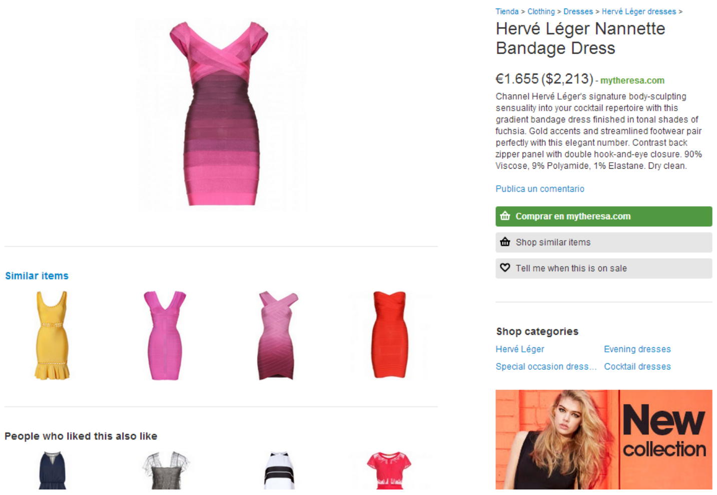


FIGURE 3.1: Virtual wardrobe

Figure 3.1 shows a screen shot of the virtual wardrobe, at the left side appears the categories of the garments. At right side the garments belonging to a particular category. Figure 3.2 shows a screen shot of a garment page, with the garment image at the left


Tienda > Clothing > Dresses > Hervé Léger dresses >

Hervé Léger Nannette Bandage Dress

€1.655 ($2,213) - mytheresa.com

Channel Hervé Léger's signature body-sculpting sensuality into your cocktail repertoire with this gradient bandage dress finished in tonal shades of fuchsia. Gold accents and streamlined footwear pair perfectly with this elegant number. Contrast back zipper panel with double hook-and-eye closure. 90% Viscose, 9% Polyamide, 1% Elastane. Dry clean.

FIGURE 3.2: Page of a garment, with its picture, textual description and category

and the textual description and category at the right (see figure A.1 for the full list of categories supported for the recommendation system).

Polyvore offers a tool to combine garments from the wardrobe, plus other decorative elements and clothing accessories, to create outfits (see figure 3.3). The outfits created with that tool are visible to the whole community and other members can visit their page ( see figure 3.4), vote on them (like them) and leave comments.

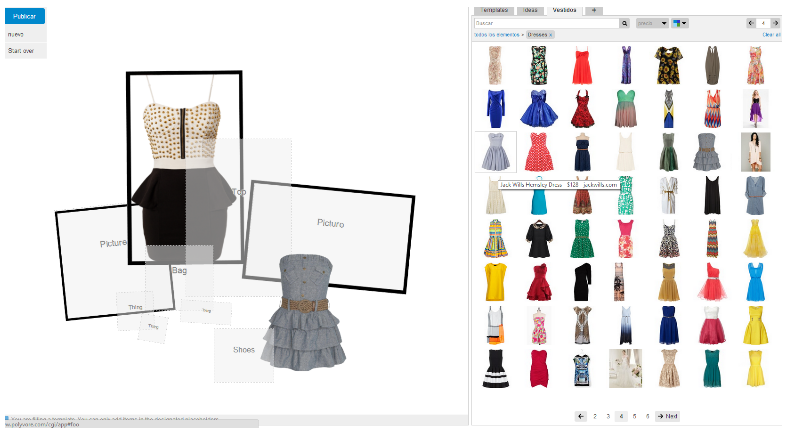


FIGURE 3.3: Outfit creation tool

Occasionally a contest is created where the goal is to craft the best outfit given an inspiration topic (i.e. summer, movies) and/or some garments restrictions (i.e. only garments from the new collection of a certain brand). Those contest are open to all the members of the community who can upload their outfits and vote for other ones. When the contest is over, the wining outfits are those with the highest number of votes. Apart from the contest prize (i.e. clothes, money,...), the winners are awarded with trophies.

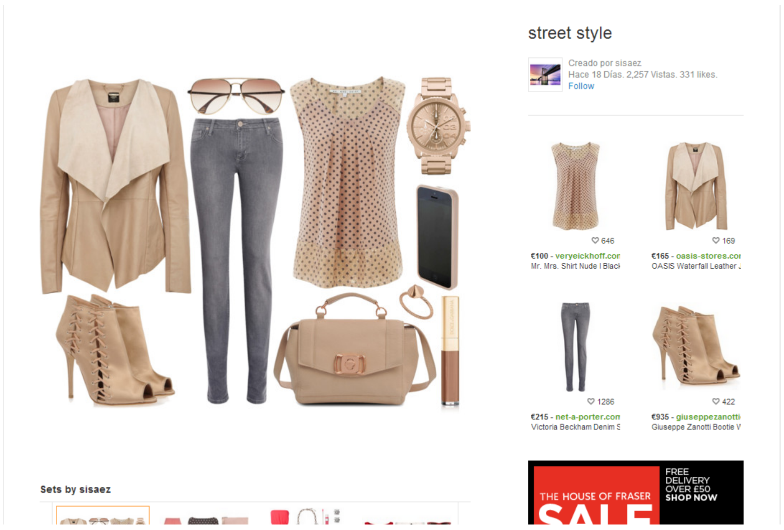


FIGURE 3.4: Outfit page

Each member of the community has its own profile where all his contributions to the community can be seen. There is listed, among others, the outfits he has created, the outfits he has voted for, the members this member follows and the ones following him.

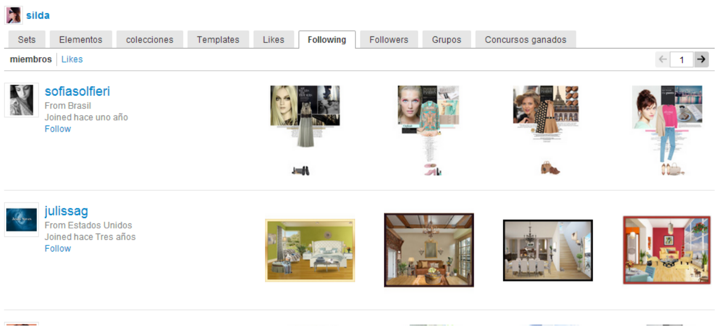


FIGURE 3.5: Member profile page with the list of outfits created by him, outfits liked, following...

Given the high user participation allowed by the site, and the highly active community behind Polyvore is a great source from where to gather valuable information from real fashion lovers.

# Chapter 4

# Design

This chapter provides a description of the design of the recommendation system. First a brief description of the whole system is given. Then the modules that conform it are presented, namely the data gathering, garment similarity, garment clustering and recommendation modules. Also, their interaction is described. After that, a more detailed description is given for each module.

## 4.1 Design overview

The initial approximation to the recommendation system was as follow:

When presented a query from a user about a garment, the system would search for all the outfits that had that particular garment. Those outfits would then be scored based on the likelihood of fitting the user taste. Finally one of those outfits would have been chosen applying a roulette algorithm.

When analysing the data from Polyvore, we found a problem with the sparsity of the data. The number of times a certain garment is part of an outfit is low (about 1.5 times). The sparsity is even bigger within the outfits created or liked by the user. This means that, when making a recommendation for a certain garment, there were not many eligible outfits.

To overcome this problem, we propose the generalization of the data. By simply observing the different garments, one can see that some of them are very similar. Some garments share the same shape, color, pattern, etc. We propose to cluster all the similar garments, those clusters are what we call abstract garments. If then we replace the original garments from the outfits by their abstraction, we will find that some of these outfits are composed by the same abstract garments. The outfits composed by abstract garments are what we call abstract outfits. One abstract outfit is composed by all the concrete outfit that, after replacing its garments for abstract garments, have all those abstract garments in common.

This generalization process allows us to reduce the data sparsity. Artificially increasing the number of times the garments appear in outfits. This leads to a larger number of possible outfits for a certain garment.

When answering a query with a garment and an user, the recommendation system collects all the abstract outfits with the abstraction of the query garment. For each abstract outfit it computes a score indicating the relevance of that outfit to the user. One of those abstract outfits is selected based on its score. A concrete outfit is selected at random from the list of concrete outfits that compose the selected abstract outfit. Finally, the query garment replaces one of the selected outfit garments with the same category. The answer given to the users is this concrete garment.

The proposed recommendation system has been split into four different modules that work together to fulfil the recommendation task. Each module performs an specific task and interacts with one or more modules. The modules are:

**Data gathering module** This module is an interface of Polyvore, its purpose is to gather all the required data for the other modules. It exposes methods to gather members, outfits and garments from the fashion community. The data is gathered making http requests to the site and parsing the response. It also provides some utilities to "clean" the captured data.

**Garment similarity module** This module takes as input the garments gathered by the data gathering module. It processes their images and textual descriptions to compute an overall similarity index for each pair of garments. This process is done applying image processing techniques and natural language analysis techniques.

**Garment clustering module** This module takes as input the data from the garment similarity module and, applying clustering algorithms, clusters the similar garments together. This clusters are what we call abstract garments. Replacing the garments from the outfits by their abstraction, a set of abstract outfits is created.

**Recommendation module** This module is the core of the recommendation system. When presented a query with a garment and an user, computes a score for every abstract garment with the abstraction of the query garment. The abstract outfit with the best score is then translated back to a concrete outfit and returned as the answer to the query.

## 4.2 Recommendation process overview

In this section, the recommendation process is described in depth.

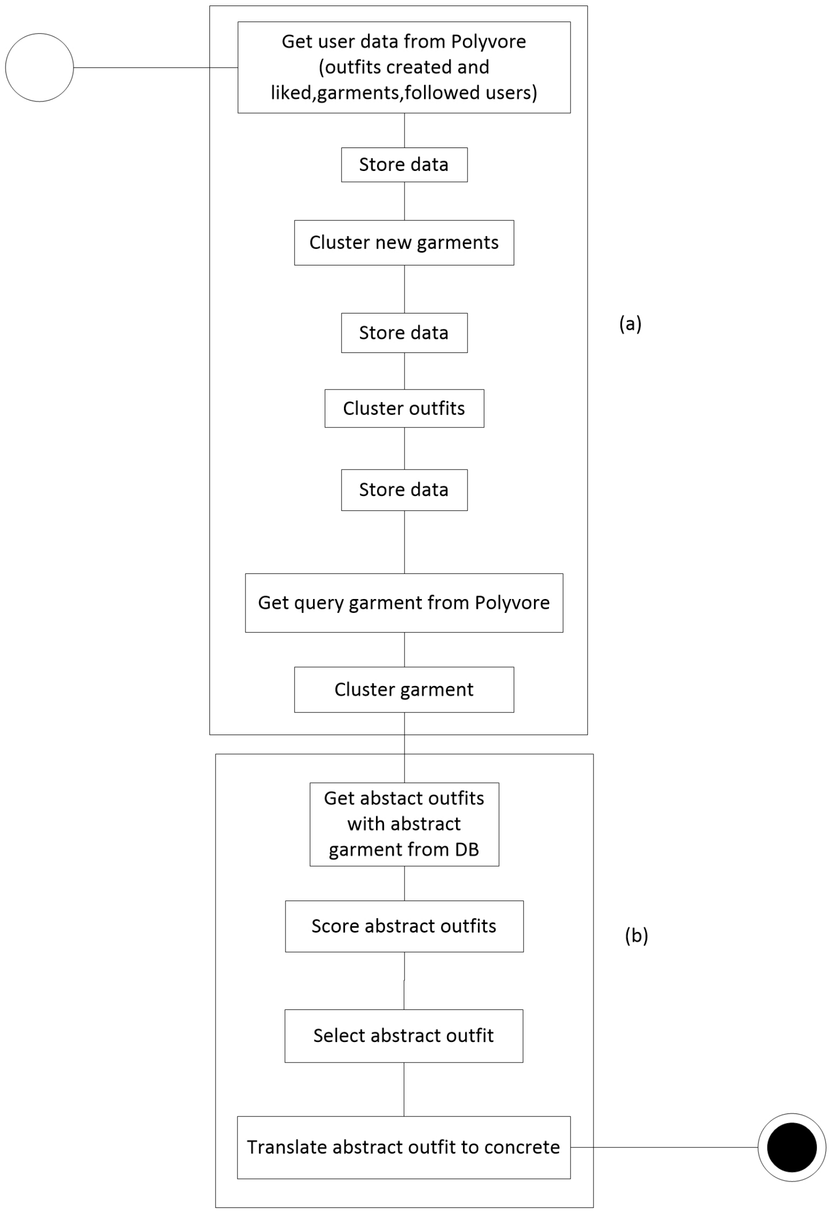

Get user data from Polyvore (outfits created and liked,garments,followed users)
Store data
Cluster new garments
Store data
Cluster outfits
Store data
Get query garment from Polyvore
Cluster garment
(a)
Get abstact outfits with abstract garment from DB
Score abstract outfits
Select abstract outfit
Translate abstract outfit to concrete
(b)

FIGURE 4.1: Recommendation process activity diagram

Figure 4.1 shows all the steps made by the recommendation system to answer a query. The process starts with a query containing the id of the garment that the users wants to combine, and the id of the user that makes the query. The subsequent steps to be taken are divided into two groups. First all the data related to the garment and the user is captured from Polyvore($a$). Once the system has all the necessary data, the recommendation process takes place ($b$).

To be able to provide a good recommendation to the user, the system needs data about his taste and also data about the garment he wants to combine. The process of gathering data takes care of this necessity.

First, all the data related to the user is captured from Polyvore. The captured data contains the outfits he has created, the outfits he has liked and the garments that compose those outfits. Additionally, the system captures the members this user follows, as well as the outfits they have created and liked, and the garments from those outfits. All this data is stored in the database for future reference.
As explained before, the recommendation process is made using abstract outfits rather than concrete ones. For this reason, the new data captured (the outfits and garments that weren't in the local database) need to be clustered.
Ideally, the system should re-cluster all the data, delete the abstract outfits and abstract garments and execute the clustering process again. This isn't done for two reasons:

- The process of clustering the garments is computationally expensive.
- The ratio between recently captured garments and garments already in the database is small, which would result in small differences between the old clusters and the new ones.

Instead, the most suitable abstract garments is assigned to each new garment. Then the abstract outfits are recalculated.
The last step of the data capturing phase of the recommendation process, is to gather the query garment from Polyvore. This new garment also needs to be clustered.
After all the necessary data is gathered and processed, the recommendation step takes place.
In this step all the abstract outfits that contain the abstraction of the query garment are scored. After scoring all that abstract outfits, one of them is selected using a roulette algorithm. Then, from the collection of concrete outfits that belong to the selected abstract outfit, one is selected at random. This outfit is modified, replacing one of its garments, that has the same category of the query garment, for the query garment. Finally, the concrete outfit is returned as answer to the query.

## 4.3 Data gathering module

The recommendation system relies on a social network as the source of data for making the recommendations. This module is an interface of Polyvore, providing a set of

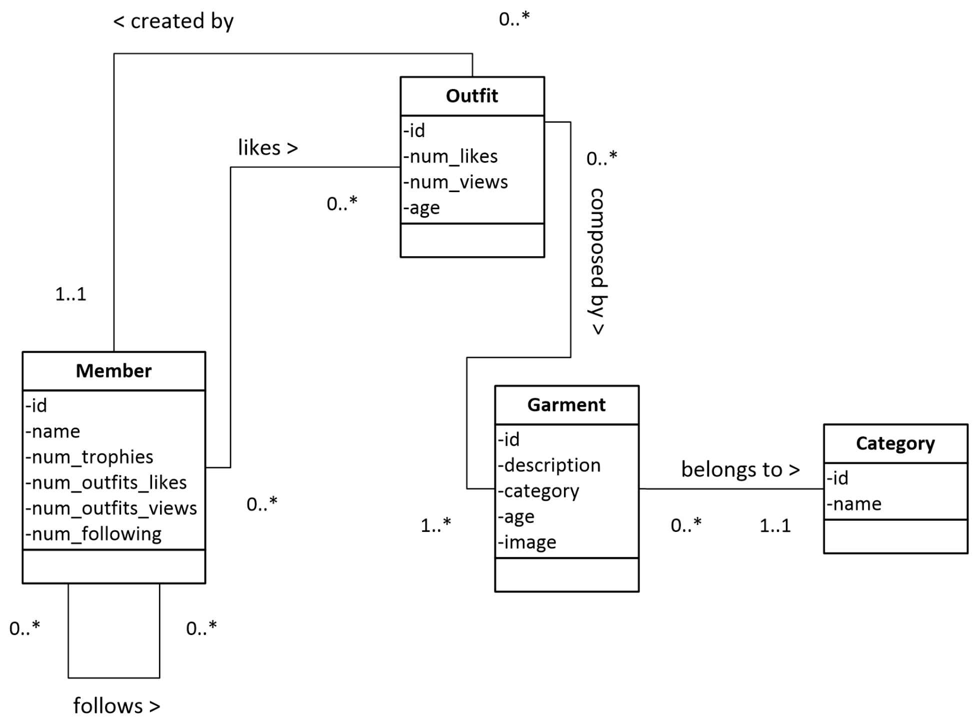


Figure 4.2: Captured objects and their relations

functions that query the site for data. This data is later used for the rest of the recommendation system. In particular, the captured data contains the following objects: members, garments and outfits and their relations.

Figure 4.2 shows a diagram of the captured objects and their relations. From the the object `Member`: the attribute `num_trophies` is the number of trophies the member has, `num_outfits_likes` is the real number of members that like an outfit created by him, `num_outfit_views` is the number of members that have seen an outfit created by him and `num_following` is the real number of members that are following him. From the object `Outfit`: the attribute `num_likes` is the real number of members that like the outfit and `num_views` is the number of members that have seen the outfit.

Given that we don't have full access to the Polyvore database, and we are just working with a subset of their data[1], the aggregation attributes (`num_outfits_likes` and `num_following` from Member and `num_likes` from Outfit) might be inconsistent with the captured data. For instance, the attribute `num_likes` of a captured outfit can be bigger than the number of captured members that have liked that outfit. Note that we used the term "real" when describing those attributes to stress that this data is coming from Polyvore and it reflects the real values for those attributes.

[1]At the beginning of the project, we sent an e-mail requesting access to Polyvore database. We explained the recommendation system that we were developing and asked for permission to access their data for research purposes. As of September 2013 we have not received an answer.

Since Polyvore has no public API available to access its data, we have developed a set of scripts to automatically gather it. Through a series of HTTP requests and the parsing of the responses, the scripts are able to retrieve the objects listed above.

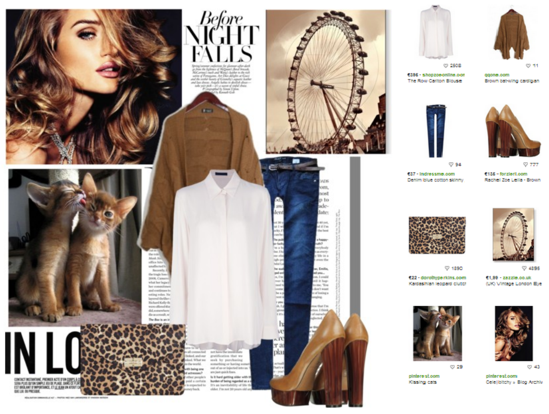


FIGURE 4.3: Outfit with the list of garments that compose it

When capturing the data associated with an outfit, we apply a filtering process to eliminate all the unnecessary data. An outfit is composed by garments, accessories and other elements that decorate the composition (letters, portraits, flowers,....). Even though all those elements configure the outfit, and the score of the outfit surely is influenced by them, our recommendation system is only prepared for dealing with garments. For this reason, any outfit component that does not belong to a category that we support (see appendix A), is not captured. For instance, figure 4.3 shows an outfit composed by three garments, a pair of shoes, a purse and other decorative element (images of a newspaper, a cat, etc). From that outfit, only the three garments will be captured.

For convenience and better interoperability between the data capturing scripts, and also the other modules, the data captured by those scripts is stored in CSV files (see section 5.4).

## 4.4 Garment similarity module

The goal of this module is to compute an overall similarity index for each pair of garments within the same category. We define the overall similarity index as a weighted aggregation of the similarity indexes with respect to the attributes observed for those garments.

In the fashion world many attributes are used to provide a precise characterization of a garment. Given the reduced data set we have for the garments, the automation of the detection of those attributes is beyond the scope of this project. To simplify the problem, we have reduced the observed attributes to a subset. The observed attributes are:

**For all types of garments:** colour, pattern, shape, fabric and category

**For tops:** neckline and sleeves

**For bottoms:** hemline

As described in the previous section, the data relative to the garments gathered from the social network consists only in a category, a picture and textual description. To be able to compare the garments in terms of the attributes recently enumerated, the value for those attributes needs to be computed. We propose to compute those values from the image of the garments and/or from the textual description. This computation is performed applying image processing and natural language analysis techniques.

Even though most of the observed attributes values can be easily extracted from the image of a garment by humans, is hard to automatize its computation. This is due to the way the pictures are taken. For instance, when searching the type of sleeves a top has, in certain pictures the sleeves can be close to the body, folded at the middle, perpendicular to the body or quite angled. This difficulty also arises when detecting the neckline, hemline, fabric and shape. As a consequence, some of the observed attributes values are only computed from the textual description and for some other only an approximation is used (see section 4.4.1.2).

Once the values for the observed attributes are computed, a similarity index with respect to each attribute from different garments needs to be obtained.

In what follows, we present the different techniques used to compute values for the observed attributes. For each attribute we also describe the similarity function used to compare two garments.

### 4.4.1 Image processing

Image processing techniques are applied to compute the similarity index with respect to colour, pattern and shape. In what follows, we present the techniques used to compute those similarity indexes.

#### 4.4.1.1 Colour similarity

To analyse the similarity between two garments with respect to their colours, the histogram of both garments is computed. Both histograms are compared using a distance function to get the similarity index.

The histogram of an image represents the provability distribution of each color from a certain color model to appear in the image. The original images are represented in the RGB color model using the cubic representation. Before computing the histograms from those images, they are translated to the HSV representation of the RGB color model. HSV divides each color into 3 separated components:

**Hue (H)** A value from 0 to 360 representing all the possible pure colors.

**Saturation (S)** A value form 0 to 100 representing the amount of white in the mix. A pure color without white is fully saturated while a color mixed with white is less saturated.

**Value (V)** A 0 to 100 value representing how much light is reflected by the color. A color with value 0 don't reflect any light and is perceived as black, while a color with value 100 reflects all the light.

This translation is done because the former representation (HSV), is known to preserve better the perceptual similarity between colors. That is, two colors close in the representation are perceived as similar by human.[15].

The histogram is computed over the H and S components of the HSV image, and the values are quantized to 30 and 32 levels (bins) respectively to reduce the computational complexity.

The similarity index between the images is computed applying a distance function to their histograms. Among all the possible distance functions, we have experimentally concluded that the correlation [26] gives us the best results. The correlation between two histograms can be expressed as:

$$d(H_1, H_2) = \frac{\sum_I (H_1(I) - \bar{H}_1)(H_2(I) - \bar{H}_2}{\sqrt{\sum_I (H_1(I) - \bar{H}_1)^2 \sum_I (H_2(I) - \bar{H}_2)^2}}$$

where $\bar{H}_k = \frac{1}{N} \sum_J H_k(J)$ and $N$ is the total number of histogram bins.

The background of the images takes more than 30% of the image area, and does not give any information of the similarity of the garments in terms of colors. Computing the histogram including the background produces histograms with very high provability for white color, reducing the provability of the colors used in the garment. This results in histograms with similar colors distributions. To avoid that, a mask that excludes the background is computed. The black area of the mask is defined by all the white color pixels ($(250, 250, 250)RGB \pm 5$). This mask is used to exclude the background area of the computation of the image histogram.

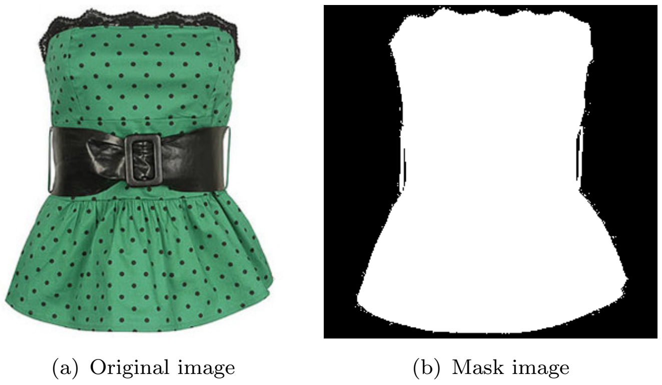

(a) Original image (b) Mask image

Figure 4.4: Original image and mask image of a garment

Figure 4.4 shows the original image of a garment and the image mask, black pixels in the image mask cover the background while white pixels cover the garment area.

(a) (b) (c) (d)

Figure 4.5: Original image and edges image of two garments

| | a | b | c | d |
|---|---|---|---|---|
| a | - | 79.03 | -0.81 | -1.17 |
| b | 79.03 | - | -0.88 | -1.50 |
| c | -0.81 | -0.88 | - | 47.09 |
| d | -1.17 | -1.50 | 47.09 | - |

TABLE 4.1: Similarity matrix of histograms of garments from figure 4.5. The values have been multiplied by 100.

Table 4.1 shows the similarity matrix for the histograms of figure 4.5. As expected, there is a strong correlation between the two pink garments ($a$,$b$) and between the two dark ones ($c$,$d$), but the correlation between the pink garments and the dark ones is much lower.

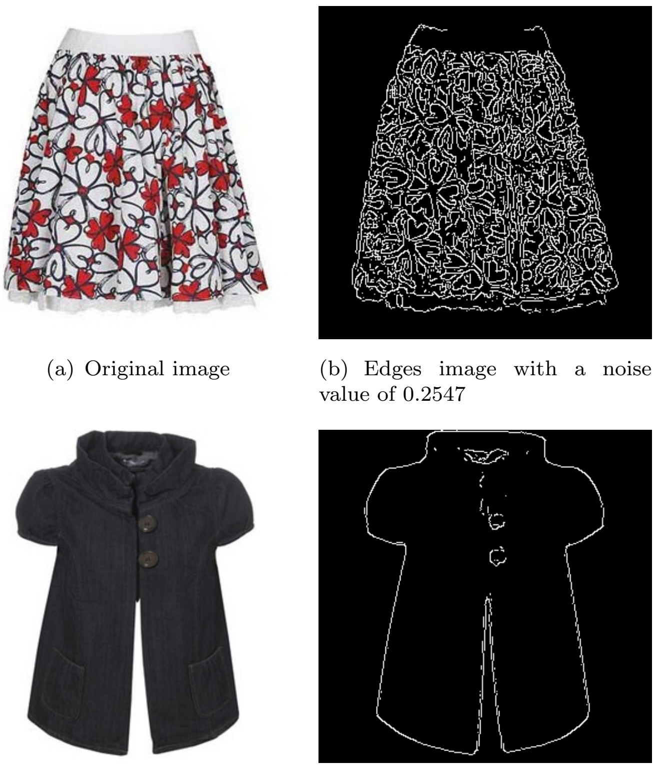
#### 4.4.1.2 Pattern similarity

There exists a vast number of different patterns: floral, dotted, animal print, stripped, etc. The task of recognizing all those patterns and computing their similarity is far beyond the scope of this project. As an alternative, we propose the measurement of noise as a naive and less accurate metric to get some information about the similarity between patterns.

We define the garment noise as the provability of a pixel of the garment image to belong to a garment edge. An edge is a point in an image where there is an accentuated change in the brightness of its surrounding pixels. Applying the Canny edge algorithm [5] to a gray scale image of the garment, a black and white image is obtained, where white pixels represent edges. Therefore, the value of noise is the provability of a pixel to belong to an edge, i.e., the provability of a pixel to be white

With this new metric, the noise for garments with intricate patterns or graphical elements will be higher than the noise for plain coloured items.

Figure 4.6 shows a comparison of the edges image generated for two different garments. The left column shows the original images and the right column shows the edges image of those garments. As can be seen, the garment in the figure ($a$) presents an intricate floral pattern. Its edges image ($b$) outlines that pattern, producing an image with a higher noise, 25.47% image (excluding the background) is white. In contrast, the garment in figure ($c$) has no pattern, only a plain color. For this garment, the edges image ($d$) only outlines the contour of the garment and few details, only 3.47% of the edges image (excluding the background) is white.



(a) Original image

(b) Edges image with a noise value of 0.2547

(c) Original image

(d) Edges image with a noise value of 0.0347

FIGURE 4.6: Original image and edges image of two garments

The similarity index for a pair of garments with respect to their pattern is:

$$1 - min(1, \frac{(noise_a * 100 - noise_b * 100)^2}{z^2})$$

Where $noise_a$ and $noise_b$ are the noise values of the garments. The constant $z$ is the maximum noise dissimilarity allowed, the value that maps to a similarity of 0. We have experimentally concluded that the value for $z$ should be 20.

#### 4.4.1.3 Shape similarity

We define the shape similarity as the accumulated similarity of the ratio width/height between both garments in each point in the garment. That is, each point between the top of the garment and its bottom. We have chosen to compare the proportions rather than the widths of the garments because the first measure is invariant to scale. The similarity function can be expressed as:

$$sim(g_1, g_2) = \sum_{0 \leq y \leq 100} 1 - min(\frac{((\frac{(W_{g_1}(y)}{h_{g_1}} * 100) - (\frac{W_{g_2}(y)}{h_{g_2}} * 100))^2}{z^2}, 1)$$

Where $W_{g_1}(y)$ is the width of the garment $g_1$ at the $y\%$ of its height. The term $\frac{W_{g_1}(y)}{h_{g_1}}*100$ represents the proportion between the width of a garment at a point and its total height $h_{g_1}$ (between the garment top and bottom). This value usually ranges between 0 and 100, given that the garments normally are wider than taller. The squared dissimilarity between the garments width at a certain point is expressed as $((\frac{(W_{g_1}(y)}{h_{g_1}}*100)-(\frac{W_{g_2}(y)}{h_{g_2}}*100))^2$. This value has been squared to stress the difference in proportions as the value grows. A dissimilarity index ranging from 0 to 1 form the dissimilarity index of the garments at a certain point (the squared difference of those garments) is computed with $min(\frac{((\frac{(W_{g_1}(y)}{h_{g_1}}*100)-(\frac{W_{g_2}(y)}{h_{g_2}}*100))^2}{z^2},1)$ where $z^2$ is the maximum dissimilarity permitted, the index of dissimilarity that maps to 1. We have experimentally concluded that the value of $z$ will be 15.

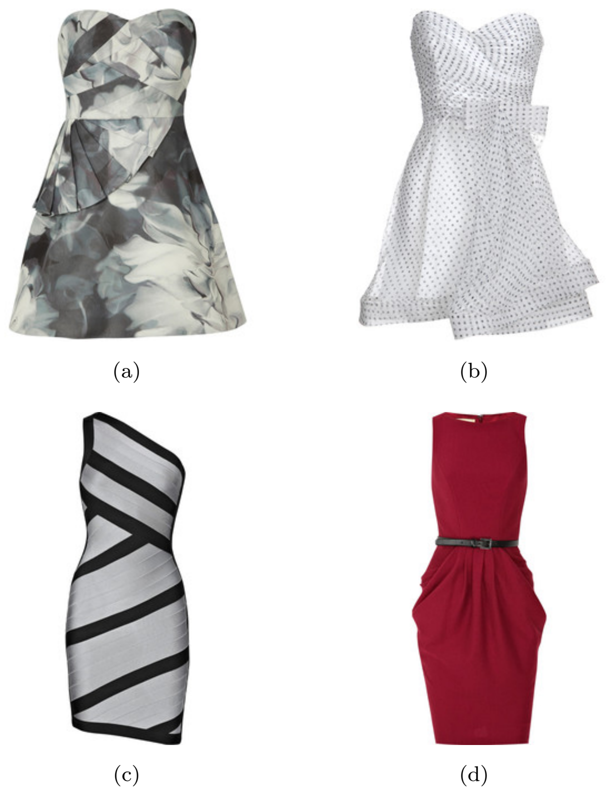


FIGURE 4.7: Images of garments

| | a | b | c | d |
|---|---|---|---|---|
| a | - | 88.61 | 33.15 | 17.15 |
| b | 88.61 | - | 34.05 | 49.32 |
| c | 33.15 | 34.05 | - | 89.55 |
| d | 17.15 | 49.32 | 89.55 | - |

TABLE 4.2: Similarity matrix of histograms of garments from figure 4.5. The values have been multiplied by 100.

Table 4.2 shows the shape similarity indexes of each pair of garments from figure 4.7. As can be seen, the pairs of garments ($a$,$b$) and ($c$,$d$) have a much higher similarity index than the pairs ($a$,$c$), ($a$,$d$), ($b$,$c$) or ($b$,$d$). This similarity indexes reflect the fact that the former pairs are much more similar in shape than the later ones.

### 4.4.2 Description processing

The main goal of description processing is to compute values for attributes that are not easily observed by the image processing module. Those attributes are the hemline, type of sleeves, neckline, fabric, shape and pattern.

The textual description of the garments is a short text with no more than 200 words written in natural language, full of specific terminology related to the fashion industry. The natural language origin of those textual descriptions gives place to the usage of synonyms and similar expressions to express the same facts. For example:

> Black silk print top with fan pleat. The fan pleat sits on the right hand shoulder. **Short sleeves**. The hemline falls to the hips. A zip fastening runs down the back. A stylish top that can be worn day or night with skinny trousers and towering heels. Pleats are on trend.
>
> **Short-sleeve** dolman tee with banded bottom and spot-print graphic detail.
>
> Brown **short sleeved** Bottega Veneta polo neck. The hemline falls to the hips. A chic knit for fall, this Bottega polo neck in an autumnal tone, is the perfect piece to team with a pencil skirt or cropped trousers and heels for a smart daywear look.

This three paragraphs show the textual description of three different garments. All those garments have short sleeves, but in each textual description, the expression used to describe that fact is different (short sleeves, short-sleeve and short sleeved respectively).

To overcome the ambiguity of the human language, a dictionary of synonyms is defined for each searched term. Figure 4.8 shows a snippet of the synonyms dictionary of the term ”one quarter length sleeve”. When processing the textual descriptions, any time this term appears or any of its synonym, the garment will be given the value associated with the term for that attribute.

```
<value name="one quarter length sleeve" id="5">
    <synonym name="1/4 length sleeve"/>
    <synonym name="1/4 sleeve"/>
    <synonym name="short sleeves"/>
    <synonym name="short sleeved"/>
</value>
```

Figure 4.8: Snippet of the attributes synonym dictionary

The similarity index with respect to an attribute observed in the textual description is as follows: if both values are the same, the similarity index is 1, if they are different, the similarity index is 0.

## 4.5 Garment clustering module

As explained in the introduction of this chapter, the reduced amount of times that a certain garment appears in outfits poses a problem when making recommendations. To overcome this difficulty we use a clustering algorithm with a similarity function based on the attributes described in the previous section. The goal of this module is to cluster the original data to reduce its sparsity.

Using the similarity indexes given by the garment clustering module (see section 4.4), a similarity matrix per category is built. The overall similarity index for two garments is a weighted aggregation of different similarity indexes, one for each observed attribute: colour, pattern, shape, fabric, neckline, sleeves and hemline.

Using a spectral clustering algorithm, the garments belonging to each category are clustered. Those cluster are what we call abstract garments. Once the abstract garments are computed, the abstract outfits are created.

Spectral clustering is a clustering technique which relies on the eigenstructure of a similarity matrix to partition elements into disjoint clusters. The resulting clusters have the property that elements in the same cluster have high similarity and elements in different clusters have low similarity.

The similarity functions described in the previous section will be used to produce a 3D similarity matrix ($m_{3D}$). This matrix has a size of $n * n * k$ where $n$ is the number of garments and $k$ is the number of similarity functions used to compare the garments. To obtain a 2D similarity matrix ($m_{2D}$) for the clustering process, the 3D matrix is reduced to a 2D matrix of $n * n$ whose value $m_{2D}[i][j]$ is the result of applying

| | $g_1$ | $g_2$ | $g_3$ | $g_4$ | $g_5$ | $g_6$ |
|---|---|---|---|---|---|---|
| $o_1$ | X | X | X | | | |
| $o_2$ | | | | X | X | X |
| $o_3$ | X | | X | | | |

(a) Initial data

| | | | |
|---|---|---|---|
| $g_1^a$ | : | $g_2$ | $g_4$ |
| $g_2^a$ | : | $g_3$ | $g_5$ |
| $g_3^a$ | : | $g_1$ | $g_6$ |

(b) Abstract and concrete garments relation

| | $g_1^a$ | $g_2^a$ | $g_3^a$ |
|---|---|---|---|
| $o_1$ | X | X | X |
| $o_2$ | X | X | X |
| $o_3$ | | X | X |

(c) Initial data having replaced the concrete garments for the abstract ones

| | $g_1^a$ | $g_2^a$ | $g_3^a$ |
|---|---|---|---|
| $o_1^a$ | X | X | X |
| $o_2^a$ | | X | X |

(d) Duplicated outfits joined into abstract outfits

Figure 4.9: Process of obtaining abstract garments and abstract outfits

$$\sum_{0 \leq l < k} (m_{3D}[i][j][l] * w[l])$$

Where $w$ is a positive integer array that sums to 1. $w[l]$ is the specific weight (between 0 and 1) given to that particular similarity function.

Figure 4.9 shows the different steps of the clustering process. Sub-figure $a$ is a representation of the original data where each outfit, denoted by $o_i$ has many garments, denoted by $g_j$.

After applying the clustering process described before, the initial garments are clustered. The resulting clusters, denoted by $g_k^a$, are what we call abstract garments. Sub-figure $b$ shows which garments belong to each of the abstract garments obtained after the clustering of the original data.

Sub-figure $c$ shows the original data after replacing the outfits with their abstraction. As can be seen, some of the outfits share exactly the same abstract garments, and therefore, are the same. For instance, $o_1$ and $o_2$ have all their abstract garments in common ($g_1^a$,$g_2^a$,$g_3^a$).
The outfits that share the same abstract garments are grouped together (sub-figure $d$). Those groups, denoted by $o_l^a$, are what we call abstract outfits.

## 4.6 Recommendation module

The recommendation module is the core module of the recommendation system. Its goal is to create an outfit with the provided garment. The resulting outfit should fit the user taste.

Given a query from an user about a garment, the module analyses a set of abstract outfits that contains the abstraction of the query garment. For each abstract outfit, a score is computed using a custom defined score function. Then, one of those abstract outfits is chosen. Finally, the system, presents the user a concrete outfit (an outfit made with concrete garments) out of the chosen abstract outfit.

In what follows, the score function used to asses each abstract outfit is described. After that, the recommendation process is explained in depth.

### 4.6.1 Score function

The goal of the score function is to assess how likely an outfit is going to be liked by the user. The function must give better scores to those outfits that are going to be liked the most, and a worse score to those that are not. To this end, the proposed score function takes into account the relevance of an outfit to the user, his closest community members and the rest of the community.

The proposed function is:

$$score(o^a) = \frac{\sum_{o \in o^a} \big(\overbrace{(userLiked * w_u)}^{\text{user taste}} + \overbrace{\frac{\#followedLikes}{\#followed} * w_f}^{\text{followed members taste}} + \overbrace{(\frac{\#likes}{\#views} * min(1, \frac{\#view}{r})) * w_c)}^{\text{community taste}}}{count(o^a)}$$

The score of an abstract outfit $o^a$ is the sum of the individual scores of the concrete outfits ($o$) that conform that abstract outfit. This sum is divided by the number of concrete outfits $count(o^a)$.

The score can be divided in 3 separated scores: user taste, followed members taste and community taste.

**User taste:** The taste of the user that makes the query is introduced by the term $userLiked * w_u$. Where $userLiked$ is a binary variable with the value being 1(0) if the user has (has not) liked the outfit. The constant $w_u$ weights the importance of the user assessment of the outfit in the global outfit score.

**Followed members taste:** Although the recommender system uses the data from a large and active fashion community, and we have applied clustering techniques, the impact of the user taste in an outfit evaluation might still be low. This can be true if the user making the query does not participate very much in the social network by means of liking outfits.

One way to get more feedback from the user is by taking into account the other members he follows. When a member follows another one, we can assume that he agrees with the other member taste. The term $\frac{\#followed-likes}{\#followed} * w_f$ takes into account the data coming from the members that the user making the query follows (followed members). Where $\#followed - likes$ is the number of followed members that liked the outfit and $\#followed$ is the total number of members the user is following. The constant $w_f$ weights the impact of the followed members assessment of the outfit in the global outfit score.

**Community taste:** To get more data from the community, a term that represent the taste of the whole community has been added.

The quotient $\frac{\#likes}{\#views}$ represents the percentage of people that liked the outfit. To avoid an outfit that has been visited one time and liked one time to have a better score than one that has been visited 300 times and liked 250, the term $\frac{\#likes}{\#views} * min(1, \frac{\#view}{r})$ is introduced. This term weights the score based on the number of times the outfit has been visited. The more the outfit has been visited, the closer its score will be to the value of $\frac{\#likes}{\#views}$. The constant $r$ fixes the minimum number of times an outfit must have been visited to have the result of $\frac{\#likes}{\#views}$. The term $\frac{\#likes}{\#views} * min(1, \frac{\#view}{r})$ represents the score given by the whole community. Again, the term $w_c$ weights the importance of the community assessment of the outfit in the global outfit score.

The summation of the 3 weight parameters $w_u$, $w_f$ and $w_c$ is 1.

### 4.6.2 Recommendation process

The recommendation process starts by gathering all abstract outfits ($o_i^a$) that have the abstraction of the query garment as one of its garments. A list of pairs $(o_i^a, s_i)$ is created. For each pair $(o_i', s_i)$, $s_i$ is the score given to the abstract outfit ($o_i^a$) using the previously described score function. Then, from this list of abstract outfits and their scores, an abstract outfit is selected using a roulette algorithm.

The fitness proportionate selection algorithm, also known as roulette algorithm, is a genetic operator used in genetic algorithms for selecting potentially useful individual from a population based on their fitness value. The provability of selecting the individual

$i_i$ is:

$$p_i = \frac{f_i}{\sum\limits_{0 \leq j < n} (f_j)}$$

where $f_i$ is the fitness value of the individual $i_i$ and $n$ is the size of the population.

To select an individual, the algorithm computes a random number $r$ between 0 and 1. The selected individual is the first one that satisfies the following conditions:

$$\sum_{0 \leq j < i} (p_j) \leq r$$

and

$$i = n - 1 \quad \text{or} \quad \sum_{0 \leq j < i+1} (p_j) > r$$

This algorithm is more likely to select the individuals with a higher fitness value, but also leaves a chance for those with lower fitness values to be selected.

In our application of the roulette algorithm, the population is the list of abstract outfits that have the abstraction of the query garment as one of its components. The fitness of an individual (abstract garment) is the score given to that abstract garment using the previously described score function.

We have chosen to apply this selection algorithm because it introduces some randomness in the selection of the concrete garment that will be given as a response, whilst being strongly influenced by the score given to the outfits (the fitness value).

Once the abstract outfit is selected, a concrete outfit is chosen at random from the outfits that conform the selected abstract outfit. Then, one of the garments from this concrete outfit, that has the same category as the query garment, is replaced by the query garment. This needs to be done to ensure that the query garment is a component of the returned outfit. Finally the modified concrete outfit is returned as answer to the query.

**Chapter 5**

# Implementation

In this chapter, a description of the implementation of the recommendation system is given. First, the technologies used are listed and briefly described. Then, the general structure of the solution is presented. After that, each package that composes the solution is described in depth.

## 5.1 Technologies

During the implementation of the proposed recommendation system the following technologies have been used:

**Java** General-purpose, object oriented computer programming language developed by Sun Microsystems. This language has a syntax influenced by the C/C++ programming languages. The code developed for the recommendation system has been mainly written using this language [34], [20].

**Python** Widely used general-purpose, high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C. This language has been used to write all the scripts that gather the data from Polyvore and parse it. It was chosen for its simplicity and for allowing a rapid coding thanks to its high level syntax [36], [22].

**Hibernate** Object-relational mapping (ORM) library for the Java language, providing a framework for mapping an object-oriented domain model to a traditional relational database. Developed by Red Hat and distributed under the GNU General Public License (GNU GPL) [11]. This library is used to provide the persistence layer of the recommendation tool. Even though its usage implies a certain overhead in the storage and retrieval of the data, it has been used to cut the time spent in the development of the data persistence layer [33], [7].

**OpenCV** (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. Developed by Itseez and released under a BSD. This library is used in conjunction with its Java bindings developed by Samuel Audet and others [3], and released under GNU GPL v2. It is used in the garment similarity module to compute attributes values from the garment images. It has been chosen for its maturity and speed [27], [28].

**MySQL** open-source relational database management system (RDBMS) developed by Oracle and released under the GNU GPL v2 license. It is used in the persistence layer of the recommendation system [35], [28].

**Weka** (Waikato Environment for Knowledge Analysis) is a popular suite of machine learning software written in Java, developed at the University of Waikato, New Zealand. Weka is free software available under the GNU GPL [31]. The suite is used in the garment clustering module, in conjunction with the spectral clustering algorithm implementation by Luigi Dragone released under the GNU GPL license [8].

**Colt** Library that provides a set of Open Source Libraries for High Performance Scientific and Technical Computing in Java. Developed by the CERN who holds its copyright. This library is a dependency of the spectral clustering algorithm [6].

**libXML2** XML C parser and toolkit developed for the Gnome project (but usable outside of the Gnome platform), it is free software available under the MIT License. This library, in conjunction with the Python bindings written by Dave Kuhlman [16], is being used by the Python scripts to parse the HTML web pages from Polyvore [30].

A part from the technologies recently described, during the execution of the project, the following software has been used:

**Windows 8** Windows 8 is a version of Microsoft Windows (an operating system developed by Microsoft) for use on personal computers, including home and business desktops, laptops, tablets, and home theatre PCs. This software is released under the license Microsoft CLUF [37],[17].

**Notepad++** Notepad++ is a free source code editor and Notepad replacement that supports several languages. Running in the MS Windows environment, its use is governed by GPL License [13].

**Eclipse** Is a multi-language Integrated development environment (IDE) comprising a base workspace and an extensible plug-in system for customizing the environment. It is written mostly in Java. This software is released under a Eclipse Public License (EPL) [10].

**MySQLWorkBench** MySQL Workbench is a unified visual tool for database architects, developers, and DBAs. MySQL Workbench provides data modeling, SQL development, and comprehensive administration tools for server configuration, user administration, backup, and much more. MySQL Workbench is available on Windows, Linux and Mac OS X [1].

**Texmaker** Texmaker is a free, modern and cross-platform $\LaTeX$editor for linux, macosx and windows systems that integrates many tools needed to develop documents with LaTeX, in just one application. This tool is licensed under the GNU GPL Version 2 [4].

## 5.2 Packages overview

The implementation of the proposed recommendation system is divided into six packages:

**Core** This package contains all the classes and interfaces that implement the domain model objects. The domain model objects are: member, garment, outfit, category, abstract garment and abstract outfit. This package also contains the mapping of those classes to the data base. This package also contains other utility classes.

**dataGathering** This package contains a set of Python scripts to capture data from the social network. This package also contains a series of Java classes that allow the integration of those scripts with the rest of the system. This package is the implementation of the data gathering module (see section 4.3).

**garmentsSimilarity** This package contains all the classes to obtain each similarity index (shape similarity, pattern similarity, color similarity and description similarity) of a certain pair of garments and their overall similarity index. This package is the implementation of the garment similarity module (see section 4.4).

**garmentsClustering** This package contains the classes for clustering the garments, creating the abstract garments and creating the abstract outfits. This package is the implementation of the garment clustering module (see section 4.5).

**recommender** Package with the classes that make recommendations to answer the queries from the users. This package is the implementation of the recommendation module (see section 4.6).

**graphicalTools** Package with the implementation of different graphic tools to visually analyse and manipulate the data. These tools are created to view the garments in the system, view the outfits, create clusters, view the abstract outfits and make queries.

Except some parts of the package `dataGathering`, all the code has been implemented using Java. The rest of the code has been implemented using Python.

Having given a brief description of all the package that compose the system, in the following sections each of them will be analysed more deeply.

## 5.3 Core package

This package contains all the classes and interfaces that implement the domain model objects. This package also contains the mapping of those classes to the data base. Finally, the package also contains other utility classes.

The classes in this package are divided in the following packages:

#### rob.ors.core.model.api and rob.ors.core.model.impl

This packages contains the classes and interfaces that implement the domain model objects. The domain model objects are: member, garment, outfit, category, abstract garment and abstract outfit.

Figure 5.1 shows a class diagram with all the interfaces from the model objects, while figure 5.2 show a class diagram of all the classes that implement the model objects. As can be seen, each class from the model has its own interface and the implementation.

### 5.3.1 Persistence layer

To store the data captured from Polyvore, as well as, the data generated by the recommendation system, a persistence layer is needed.

The persistence of the recommendation system data has been implemented using a relational database. The storage and retrieval of the Java objects is managed by an ORM.

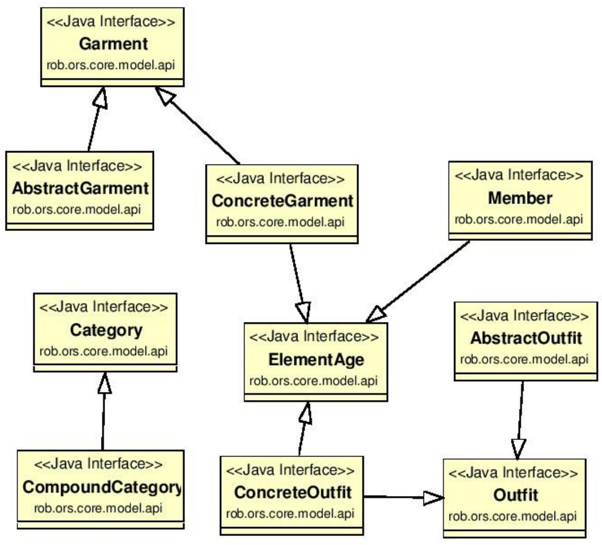


Figure 5.1: Classes diagram of the system model classes

Although using a ORM in an application such as this can introduce some overhead, we decided to use it as it reduces the time spent in the implementation of the persistence layer.

The database has been implemented using MySQL. The file, found in the root of this package, `CREATE.sql` contains a SQL script to create the database schema.

Figure 5.3 shows the diagram of the database.

The ORM being used is Hibernate. The XML files with the configuration of the mapping between the Java objects and the database tables can be found at the root of the core package. Files `AbstractGarment.hbm.xml`, `AbstractOutfit.hbm.xml`, `Category.hbm.xml`, `ConcreteGarment.hbm.xml`, `ConcreteOutfit.hbm.xml` and `Member.hbm.xml` are the configuration files for their respective Java model objects. These files contain the definitions of how the object attributes will be mapped to database table columns. File `hibernate.cfg.xml` is the Hibernate general configuration file. This file specifies the database connection parameters.

### rob.ors.core.polyvore

The class `PolyvoreCategoryTree` contains a static representation of the garment categories used by Polyvore, as well as some convenience methods for traversing it and obtaining the garments in each category.

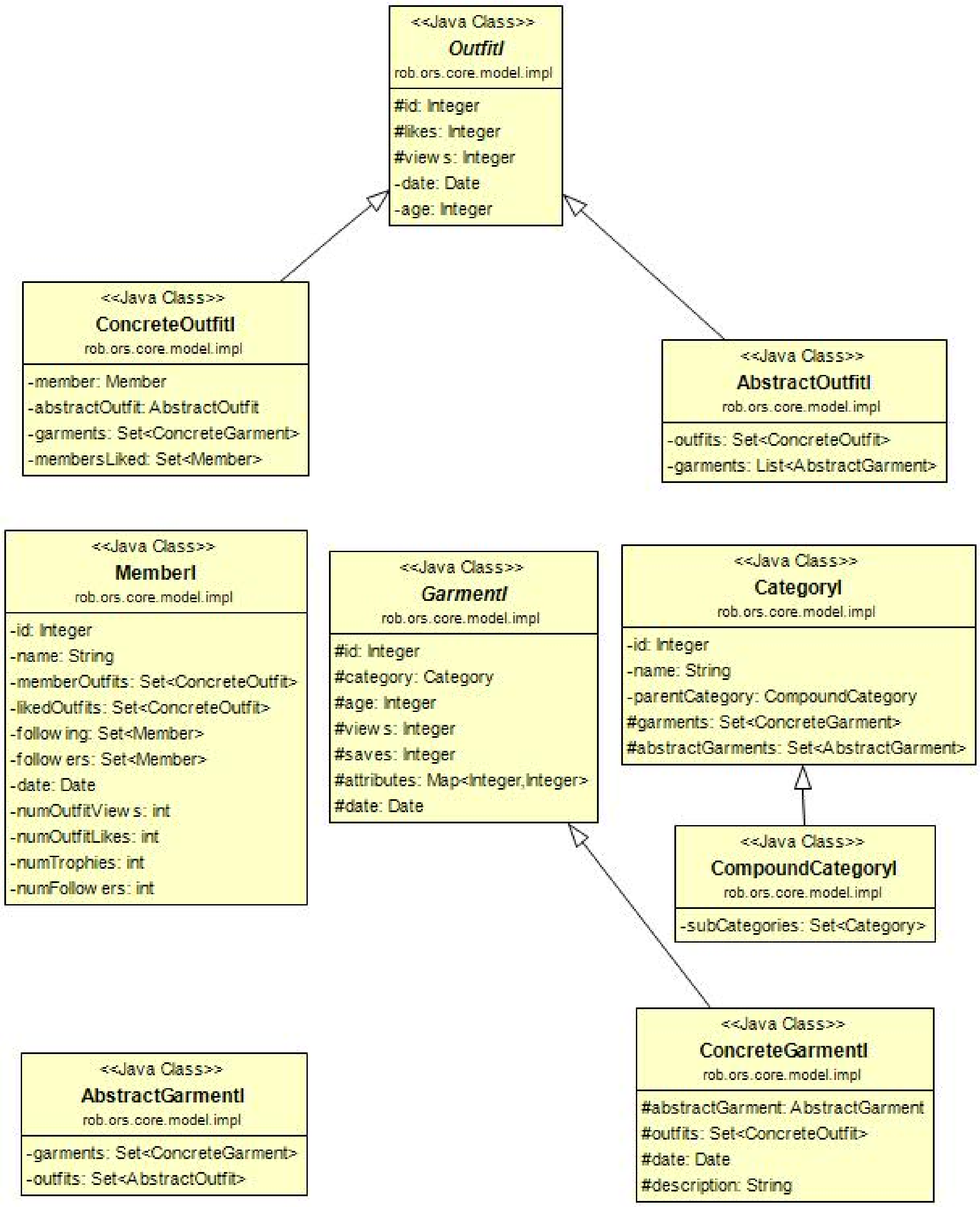


FIGURE 5.2: Classes diagram of the system model classes implementation

### rob.ors.core.utils

The class `GarmentRemover` is an utility class used to remove from the system elements that are not garments, get the ids of those elements and check if an id belongs to the list of removed elements. We implemented the later two functionalities to avoid re-capturing elements that have already been removed from the system.

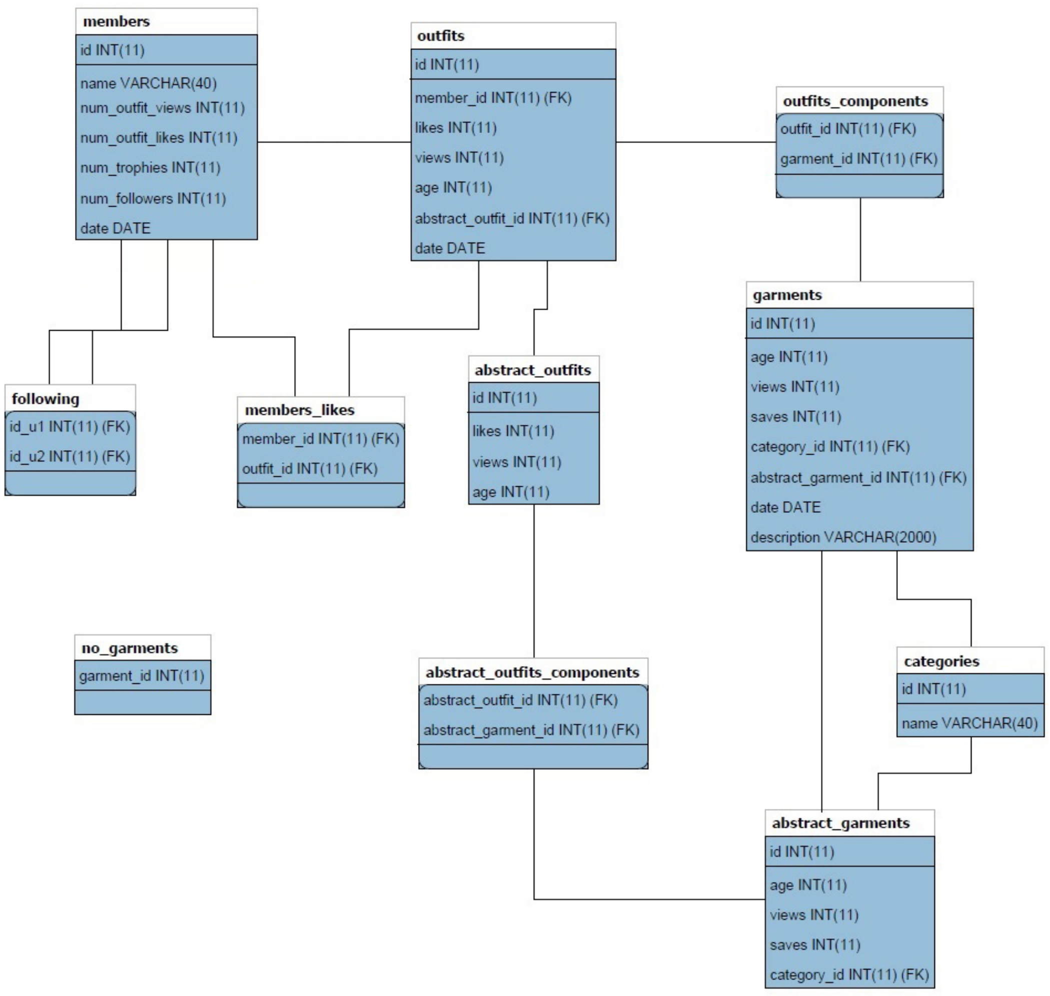


FIGURE 5.3: Database diagram

**rob.ors.core.config**

The only class contained in this package is the class `Paths`. This class has a series of static strings with the paths for various files and directories used by the rest of the code.

## 5.4 DataGathering package

This package is the implementation of the data gathering module (see section 4.3). The package contains two separated sources: the Python scripts for querying Polyvore, and a set of Java classes to integrate the Python scripts into the system. This Java classes allow the execution of these scripts directly from Java and also reading the coma separated values (CSV) [32] files created by those.

The format of the CSV files generated by the Python scripts is as follows:

**outfits.csv** Id;Age;#Views;#Likes;Garments

**garments.csv** Id;Category;Age;#Views;#Saves;Description

**members_data.csv** Id;Outfits;Liked;Following;Followers

**members_summary.csv** Id;#Outfit views;#Outfit likes;#trophies;#Followers

The CSV format has been chosen to easy the communication between the multiple data gathering scripts. Each script is in charge of gathering a certain piece of data, but it might rely on data gathered by another script. For instance, the script `getoutfits.py` used to get data to initially populate the database (explained before), reads the ids of outfits from the file `members_data.csv`, which is generated by the script `getmembers.py`.

### 5.4.1 Python scripts

The Python scripts found in the data gathering package are the following:

**getmembers.py** This script reads the file `member_summary.csv`. For each member in the file, captures and stores its data. If the file is empty, the script starts capturing the member with id 349251. The order followed for capturing the members is based on the weighted aggregation of the values #Outfit views; #Outfit likes; #trophies and #Followers. All the captured data is stored in two CSV files: `members_summary.csv` and `members_data.csv` using the CSV formats explained before.

**getoutfits.py** This script reads the CSV file `users.csv` (the output form `getmembers.py`). For each member in the file, captures all the data relative to the outfits created by that member. The captured data is stored in the file `outfits.csv` using the CSV format explained before.

**getgarments.py** This script reads the CSV file `outfits.csv` (the output from `getoutfits.py`). For each outfit in the file, captures the data of the garments that compose it. The captured data is stored in the file `garments.csv` using the CSV format explained before.

**getGarment.py, getOutfit.py, getMember.py** These scripts receive as parameter an object id (garment, outfit and member id respectively). They capture the object data and print it out in the CSV format explained before. Those scripts allow capturing garments, outfits and members directly from the Java code.

**garmentgetter.py, membergetter.py, outfitgetter.py** The goal of this classes is to capture the data for of certain object (garment, member and outfit respectively). All of them have the method `get_data`. This method receives as parameter the object id and returns a dictionary with the captured data. All those classes extend the class `CommonGetter` found in `commongetter.py`. This class provides a set of methods used to query Polyvore and analyse the HTML response.

**dataFileUtils.py** A file with common functions to read and write CSV files.

**getImage.py** A script that receives as parameter the id of a garment an downloads its image from Polyvore.

**taxonomy.py** Class that reproduces the categories used by Polyvore.

**dataAnalysis.py** Collection of scripts that provide statistics about the captured data. Among others, they compute the percentage of garment captured form the outfits created by the users, the mean number of times a garment appears in outfits, etc.

To be able to respond queries, the system needs to be populated with an initial set of users, outfits and garments. From this data the initial abstract garments and abstract outfits will be created. Also, this data is used for the community assessment of possible outfits, calculated during the recommendation process. To capture this initial data, the scripts `getmembers.py`, `getoutfits.py`, `getgarments.py` have been developed (For further information on how to populate the system, see section 7.2).

### 5.4.2 Java code

The Java code found in the data gathering package is divided in the following packages:

#### rob.ors.informationGathering.filler

This package contains 3 classes for reading data from the CSV files generated by the Python scripts and store it in the database. These classes are: `FillGarmentsFromFile`, `FillOutfitsFromFile` and `FillMembersFromFile`. These classes read the garments, outfits and members CSV files respectively and store the data in the database.

#### rob.ors.informationgathering.getters

This package contains 3 classes (`GarmentGetter`, `OutfitGetter` and `MemberGetter`) to get data of garments, outfits and members respectively, from the local database, when available, or from the web when necessary. These classes have the methods `getGarment(Integer iid, boolean update, int deepness)`, `getOutfit(Integer iid, boolean update, int deepness)` and `getMember(Integer iid, boolean update, int deepness)`. These methods receive as parameters the id of the object to capture, a boolean indicating the request for updating the local data if found, and the deepness allowed. The flag `update` indicates if the local data, if existent, must be updated. When set to true, if the requested object exists in the local database and its age is bigger than the maximum age allowed, the data will be fetched again from Polyvore. The age of an object is the number of days that have passed since it was captured from Polyvore. The maximum age allowed is set to 30 days. This technique of locally catching the data allows to reduce the request to Polyvore and makes the recommendation process faster. The deepness parameter indicates how deep the current call will go into fetching the related objects to the object being obtained. For instance, when calling `getGarment` with the deepness parameter set to 0, only the garment will be captured. For a deepness value of 1, the garment will be obtained from Polyvore and the outfits that use that garment (including the garments that compose these outfits). For a deepness value of 2, the garment will be obtained from Polyvore, the outfits that use that garment (including the rest of the garments that compose these outfits) and the outfits where the later garments are used (including the rest of the garments that compose these outfits). This technique allows to control how much data is captured from Polyvore. The deepness parameter can take any positive value or 0.

## 5.5 GarmentSimilarity package

This package is the implementation of the garment similarity module (see section 4.4). The package contains all the classes used to compute the values for all the observed attributes. The package also contains the classes used to compute the similarity indexes between garments with respect to the observed attributes. Finally, the package also contains the classes used to compute the overall similarity index between a pair of garments.

The package is divided into three packages:

#### rob.ors.garmentssimilarity.imageProcessing

This package contains the implementation of the image processing techniques described in the sub-section 4.4.1. The package contains all the classes for computing the values of all the attributes observed from the garment images. These attributes are: colour, shape and pattern. This package also contains a class to extract the background mask from a garment.

The classes found in this package are:

`BackgroundExtractor` This class has the method `colorMaskIpl(IplImage src)`. This method receives as a parameter a garment image and returns the image of the garment mask.

`HistogramExtractor` This class has the method `CvHistogram[][] getHSPatchHistogram( IplImage src, CvMat itemMask, float relativePatchSize)`. This method receives as parameters a garment image, its mask and the relative image patch size. An image patch is defined as a region of the said image [1]. This method returns an array of histograms for the garment image. The number of histograms computed depend on the parameter *relativePatchSize*. This parameter, with a value ranging from 0.1 to 1, indicates the size of the patches into which the image is divided. A value of 0.5 means that the patches will have a size of half of the image, that is, the image will be divided in 4 patches. The return of the method is an array of histograms, one fore each patch.

`ItemProportionsExtractor` This class has the method `float[] getItemProportions( CvMat itemMask)`. This method receives as a parameter the garment mask and returns an array of 100 positions. This array contains the values of the proportion between the garment width at every percentage of its total height (from 1 to 100, with an step of 1), and the total height.

`NoiseDetection` This class has the method `noiseValue(IplImage image,CvMat mask)`. This method receives as parameters a garment image and its mask and returns the value of the noise found in the garment.

#### rob.ors.garmentssimilarity

The only class in this package, `GarmentsComparator`, has 3 methods to obtain a similarity index for each of the observed attributes in a garment image, that is colour, shape and pattern. These methods are:

**patchHistogramSimilarity( IplImage i1,CvMat m1, IplImage i2, CvMat m2, float pSize)** This method receives as parameters two pairs (garment image, garment mask).

[1]When applying image similarity techniques based on histogram similarity, it is a common approach to divide the compared images into regions (patches). Although our implementation of the image processing module follows this approach, we have experimentally concluded that the patch size value that give us better results is 1. That is, we divide the garments images in only one patch.

The method returns the histogram similarity index of the garments. The parameter `pSize` (from 0 to 1) indicates the relative size of each patch to be compared. The final similarity index is the mean of the similarity index of each pair of histograms $histArray_{g1}[i], histArray_{g2}[i]$ where $histArray_{g1}$ and $histArray_{g2}$ is the histogram array of garment 1 and 2, respectively.

**noiseSimilarity(IplImage i1,CvMat m1, IplImage i2, CvMat m2)** This method receives as parameters two pairs ( garment image, garment mask). The method returns the similarity index between the garments based on their noise value. This value is computed using the class `NoiseDetection`.

**widthsSimilarity(float[] anItemProportions, float[] anotherItemProportions)** This method receives as parameters two arrays of garments proportions and returns the similarity index based on the shape of those garments. The arrays of garments proportions are computed using the method `getItemProportions(CvMat itemMask)` from the class `ItemProportionsExtractor`.

**rob.ors.itemssimilarity.textProcessing**

This package contains the implementation of the textual description processing techniques described in the sub-section 4.4.2. The goal of the classes from this package is to compute a value for each attribute observed in the textual description.

The class `DescriptionProcessor` has the method `getAttributesFromDescription(Garment garment)`. This method receives as parameter a `Garment` instance and assigns a value to every attribute described in the file `attribute.xml` (see appendix B). This method traverses all the file `attributes.xml`. For every `attribute` tag in that file, the method searches if the garment description has any of the possible values associated with this attribute. If found, that garment is assigned that value for that particular attribute. If none of the possible values is found, a default value is assigned.

The file `attributes.xml` is a XML file that contains, for each observed attribute from the garment description, a list of all its possible values and its synonyms. Figure 5.4 shows an example of an attribute from the file `attributes.xml`. Each attribute has a name and an id. Also a `defaultId`, that is the id of the default value that will be assigned to a garment that do not have any of the attribute possible values. Between the tags $< values >< /values >$ all the possible values for the attribute are listed. Each value has a name and id. Inside each value are defined the possible synonyms, that is, all the other expressions that map to that value. In the given example, the term "above the knee" will be mapped to the value "above knee". Finally, the definition of

```
  <attribute name="hemline" id="4" defaultId="13">
    <values>
<value name="above knee" id="5">
<synonym name="above the knee"/>
<synonym name="above the knees"/>
<synonym name="above knees"/>
        </value>
   </values>
   <automatic-synonyms>
       <append term=" length"/>
       <append term="-length"/>
   </automatic-synonyms>
</attribute>
```

Figure 5.4: Snippet of the attributes synonym dictionary

an attribute also has a list of automatic synonyms. Those are characters added to the beginning of a value or any synonym (prepend) or to the end of it (append) to create a new synonym for the value. For instance, in the given example, the automatic synonym defined as $< append\, term = "\ lengt"/ >$ in conjunction with the value *above knee* will create the new synonym *above knee lengt*. The automatic synonyms are applied to the attributes values itself or any of its synonyms.

`AttributeD` and `AttributesDictionary` are a Java translation of the `attributes.xml` file. `AttributeDictionary` has a collection of `AttributeD` which has a collection of its possible values.

## 5.6 GarmentsClustering package

This package is the implementation of the garment clustering module (see section 4.5). The package contains the classes for clustering the garments and creating the abstract garments and abstract outfits.

Figure 5.5 shows a class diagram with the most important classes from the garment clustering module. In what follows, the most relevant classes will be described.

#### CompoundSimilarityMatrix

As explained in the previous chapter, the spectral clustering technique used requires a similarity matrix of the elements to be clustered. This matrix is a square matrix ($m$) where each value ($m[i][j]$) contains the similarity value between garments $i$ and $j$. Despite that, our similarity matrix is composed by multiple slices, one for each similarity

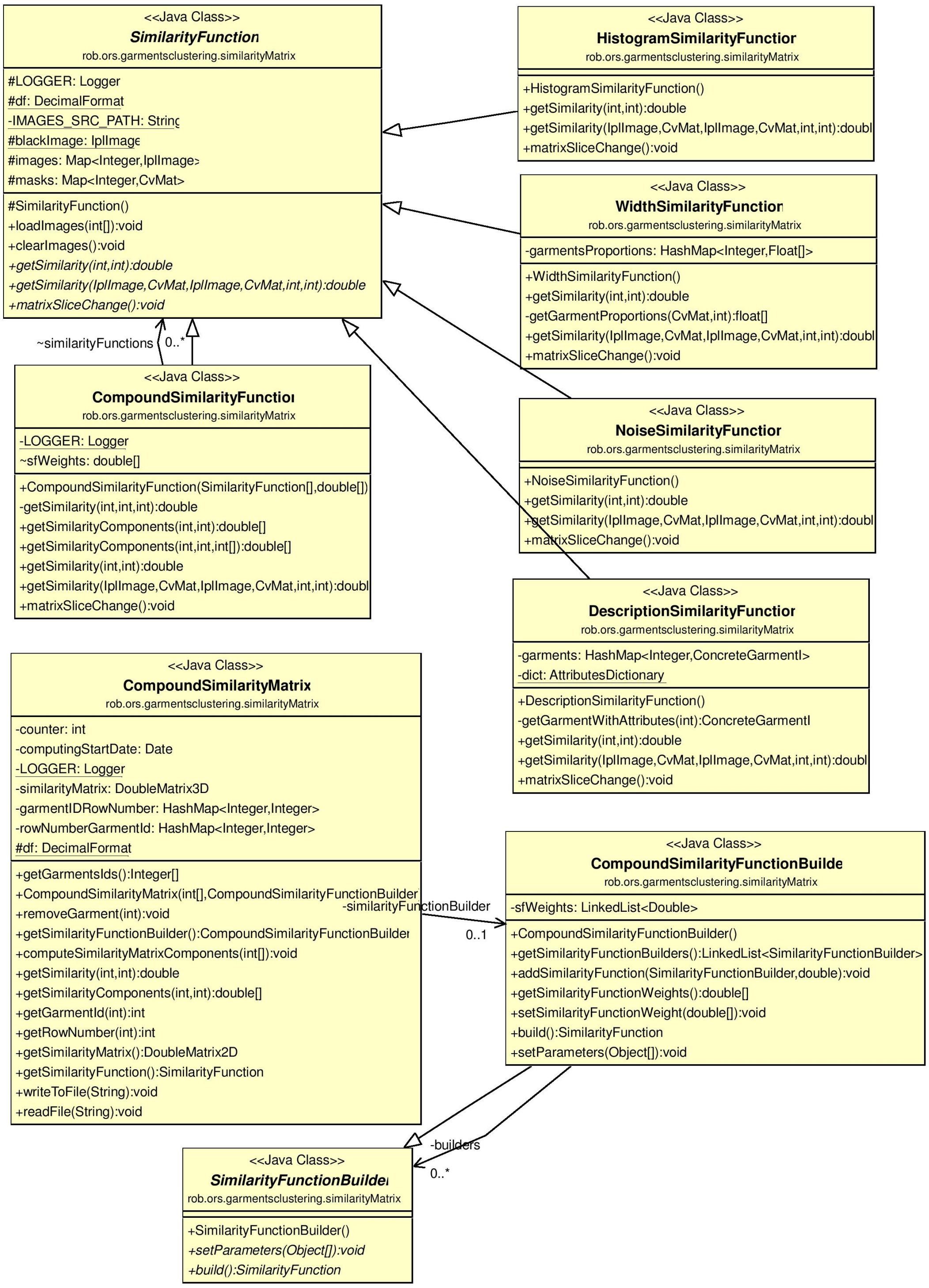

<<Java Class>>
SimilarityFunction
rob.ors.garmentsclustering.similarityMatrix
#LOGGER: Logger
#df: DecimalFormat
-IMAGES_SRC_PATH: Strinç
#blackImage: IplImage
#images: Map<Integer,IplImage>
#masks: Map<Integer,CvMat>
#SimilarityFunction()
+loadImages(int[]):void
+clearImages():void
+getSimilarity(int,int):double
+getSimilarity(IplImage,CvMat,IplImage,CvMat,int,int):double
+matrixSliceChange():void
<<Java Class>>
HistogramSimilarityFunctior
rob.ors.garmentsclustering.similarityMatrix
+HistogramSimilarityFunction()
+getSimilarity(int,int):double
+getSimilarity(IplImage,CvMat,IplImage,CvMat,int,int):doubl
+matrixSliceChange():void
<<Java Class>>
WidthSimilarityFunctior
rob.ors.garmentsclustering.similarityMatrix
-garmentsProportions: HashMap<Integer,Float[]>
+WidthSimilarityFunction()
+getSimilarity(int,int):double
-getGarmentProportions(CvMat,int):float[]
+getSimilarity(IplImage,CvMat,IplImage,CvMat,int,int):doubl
+matrixSliceChange():void
~similarityFunctions
0..*
<<Java Class>>
CompoundSimilarityFunctioı
rob.ors.garmentsclustering.similarityMatrix
-LOGGER: Logger
~sfWeights: double[]
+CompoundSimilarityFunction(SimilarityFunction[],double[])
-getSimilarity(int,int,int):double
+getSimilarityComponents(int,int):double[]
+getSimilarityComponents(int,int,int[]):double[]
+getSimilarity(int,int):double
+getSimilarity(IplImage,CvMat,IplImage,CvMat,int,int):doubl
+matrixSliceChange():void
<<Java Class>>
NoiseSimilarityFunctior
rob.ors.garmentsclustering.similarityMatrix
+NoiseSimilarityFunction()
+getSimilarity(int,int):double
+getSimilarity(IplImage,CvMat,IplImage,CvMat,int,int):doubl
+matrixSliceChange():void
<<Java Class>>
DescriptionSimilarityFunctior
rob.ors.garmentsclustering.similarityMatrix
-garments: HashMap<Integer,ConcreteGarmentI>
-dict: AttributesDictionary
+DescriptionSimilarityFunction()
-getGarmentWithAttributes(int):ConcreteGarmentI
+getSimilarity(int,int):double
+getSimilarity(IplImage,CvMat,IplImage,CvMat,int,int):doubl
+matrixSliceChange():void
<<Java Class>>
CompoundSimilarityMatrix
rob.ors.garmentsclustering.similarityMatrix
-counter: int
-computingStartDate: Date
-LOGGER: Logger
-similarityMatrix: DoubleMatrix3D
-garmentIDRowNumber: HashMap<Integer,Integer>
-rowNumberGarmentId: HashMap<Integer,Integer>
#df: DecimalFormat
+getGarmentsIds():Integer[]
+CompoundSimilarityMatrix(int[],CompoundSimilarityFunctionBuilder
+removeGarment(int):void
+getSimilarityFunctionBuilder():CompoundSimilarityFunctionBuildeı
+computeSimilarityMatrixComponents(int[]):void
+getSimilarity(int,int):double
+getSimilarityComponents(int,int):double[]
+getGarmentId(int):int
+getRowNumber(int):int
+getSimilarityMatrix():DoubleMatrix2D
+getSimilarityFunction():SimilarityFunction
+writeToFile(String):void
+readFile(String):void
-similarityFunctionBuilder
0..1
<<Java Class>>
CompoundSimilarityFunctionBuilde
rob.ors.garmentsclustering.similarityMatrix
-sfWeights: LinkedList<Double>
+CompoundSimilarityFunctionBuilder()
+getSimilarityFunctionBuilders():LinkedList<SimilarityFunctionBuilder>
+addSimilarityFunction(SimilarityFunctionBuilder,double):void
+getSimilarityFunctionWeights():double[]
+setSimilarityFunctionWeight(double[]):void
+build():SimilarityFunction
+setParameters(Object[]):void
-builders
0..*
<<Java Class>>
SimilarityFunctionBuildeı
rob.ors.garmentsclustering.similarityMatrix
+SimilarityFunctionBuilder()
+setParameters(Object[]):void
+build():SimilarityFunction

Figure 5.5: Classes diagram of garment clustering module

function used to compare the garments. The similarity matrix is implemented in the class `CompoundSimilarityMatrix`. This class internally holds a $n*n*k$ matrix instance of `DoubleMatrix3D`, where $n$ is the number of garments and $k$ is the number of slices, one for each similarity function used.

The method `computeSimilarityMatrixComponents(int[] slices)` computes the 3D similarity matrix of the garments for the specified slices (similarity functions). This method divides the matrix into smaller sub-matrices, and computes their similarity. This allows to store all the images of the garment being compared an their masks in memory, and reduce the time spent loading images from the file system. We have experimentally concluded that the sub-matrices size should be 100*100. This value could not be set to the total of the garments being clustered, as that produced out of memory errors. Those sub-matrices can also be computed in parallel, having $n$ threads each of them computing an array of sub-matrices.

Once the 3D similarity matrix $m_{3D}$ has been computed, the method `getSimilarityMatrix()` returns a 2D similarity matrix ($m_{2D}$). The matrix $m_{2D}$ of size $n * n$ is obtained from the matrix $m_{3D}$ applying the following function to each pair of indexes $i$,$j$:

$$m_{2D}[i][j] = \sum_{0 \le l < k} (m_{3D}[i][j][l] * w[l])$$

Where $w[l]$ is the specific weight (between 0 and 1) given to that particular similarity function. The summation of all the values of $w$ is 1.

With the methods `writeToFile(String)` and `readFile(String)`, the 3D similarity matrix can be stored in a file and loaded from it.

The constructor of the class takes as input the ids of the garments to cluster and a instance of `CompoundSimilarityFunctionBuilder` initialized with the appropriate similarity function builders (explained later).

### SimilarityFunction

Each similarity function must extend the abstract class `SimilarityFunction` that provides common functionalities. The class `CompoundSimilarityFunction` is a special implementation of `SimilarityFunction` that holds many similarity functions inside, each one with its specific weight.

Each `SimilarityFunction` instance is build using an special class `SimilarityFunctionBuilder`. This class allow the parametrization of the similarity function with the passed parameters. This structure has been developed to allow the parallel computation of the similarity matrix. Conversely, the class `CompoundSimilarityFunction` has associated the class `CompoundSimilarityFunctionBuilder`.

The classes `DescriptionSimilarityFunction`, `HistogramSimilarityFunction`, `NoiseSimilarityFunction` and `WidthSimilarityFunction` are concrete similarity functions that compute the similarity between garments based on their description, colour, pattern and shape respectively.

### MySpectralClustering

The class `MySpectralClustering` is the implementation of the spectral clustering algorithm, based on the one by Luigi Dragone. This class has the method `public void buildClusterer(int[] garments, CompoundSimilarityMatrix similarityMatrix)`. This method receives as parameters an array of garments ids (the training set) and a `CompoundSimilarityMatrix` initialized with the appropriate garments and an instance of `CompoundSimilarityFunctionBuilder`.

As result of this call, the cluster where any garment from the training set belongs can be obtained using the method `int getTrainingInstanceCluster(int garmentId)`. Any garment not belonging to the training set can then be clustered using the method `int clusterInstance(int garmentId)`. This method returns the cluster of the training set garment that is less distant to the garment to cluster.

## 5.7 Recommender package

This package is the implementation of the outfit recommendation module (see section 4.6). The package contains the class `Recommender`. This class has the method `Outfit makeRecommendation(Integer uid, Integer garmentId)` that receives as parameters an id of a Polyvore user and an id of a garment. The return of this method is a `ConcreteOutfit` that contains the `ConcreteGarment` identified by `gatmentId` as one of its garments.

The pseudo code for the method is as follows:

```
(1) GetMemberData(uid);
(2) queryGarment = GetConcreteGarmentData(garmentId);
(3) queryAbstractGarment = GetGarmentAbstraction(queryGarment);
(4) possibleAbstractOutfits = GetAbstractOutfitsWithAbstractGarment(queryAbstractGarment);
(5) possibleAbstractOutfitsWithScores = ScoreAbstractOutfits(possibleAbstractOutfits,member);
(6) selectedAbstractOutfit = RouletteSelectAbstractOutfit(possibleAbstractOutfitsWithScores);
(7) responseOutfit RandomSelectConcreteOutfit(selectedAbstractOutfit);
(8) responseOutfit = ReplaceWithGarment(responseOutfit,queryGarment);
(9) return responseOutfit;
```

First (1) the data for the member is obtained. If local data exists for the user and it is not older than a certain age, this data is used. If not, the recommendation system queries Polyvore for the data of the user and its closest members. This data includes the outfits created or liked by them as well as the garments that compose those outfits. After gathering the new data, the new captured garments are clustered and the abstract outfits are recreated.

Then (2), the data for the query garment is obtained. Again, if the data exists in the local database and it is not older than a certain age, this data is used. If not, the data is fetched from Polyvore.

The step (3) obtains an abstract garment out of the original query garment. If the query garment comes from the local database, the garment already has an abstract garment assigned. On the contrary, when the query garment has been fetched in the last operation, the garment needs to be assigned an abstract garment. Applying a simple clustering algorithm, the query garment is compared with every other garment in the database that already has an abstract garment. The resulting abstract garment is the one that is assigned to the garment closer to the `queryGarment`, the one with the bigger overall similarity index.

Step (4) gets a list of all the abstract outfits that have the `queryAbstractGarment` as a component. These abstract outfits are then scored using the score function (see subsection 4.6.1) (5). Applying a roulette selection algorithm (see section 4.6), an abstract outfit is selected (6). Then, at random, one of the concrete outfits that compose this abstract outfit is selected (7). Finally, from the selected concrete outfit, one of the garments whit the same category as the query garment is replaced by the query garment. The resulting concrete outfit is returned as an answer to the query.

## 5.8 GraphicalTools package

This package contains graphical tools for analysing the data, detecting elements that are not garments and clustering the garments. The package contains the following classes:

`AbstractOutfitsViewer` This class allows the user to view the abstract outfits currently in the system. This class uses the class `AbstractOutfitsPanel` to show all the outfits.

`ConcreteOutfitsPanel` This class allows the user to see the concrete outfits in the system.

`RecommendationPanel` This class creates a graphical tool that allows the user simulate queries to the system. After entering an id of a member and an id of a garment, the system will compute the possible recommendations and show them to the user.

`ClustersPanel` This class creates a graphical tool that allows the user to visualize the garments grouped by categories. It also allows the user to cluster those garments using the similarity functions already presented. With this tool, the user can specify the appropriate parameters for the clustering algorithm.

`SuspiciousGarmentsPanel` This class creates a graphical tool that traverses all the garments in the database and shows those that provably are not garments, elements like decorative images, images of garments worn by models, clothing accessories categorized as garments, etc. An element is considered as potentially not being a garment if any of the following is true:

- The difference between the maximum and minimum proportions of the garment image (being the proportion the quotient between the width at a certain point and the total height, not including the background), is less than a certain threshold.
- The image background area with respect to the total garment image area is bigger than a certain value or smaller. This detects images that cover practically all the image area, or just a small part.

`View` This class creates a frame where the already presented graphical tools are displayed.

In section 7.2, all the recently presented graphical tools are described in depth.

# Chapter 6

# Performance analysis

Some of the processes performed by the recommendation system are very computational expensive. In particular, the processes that consume the more resources are the computation of the garments similarity matrix and the clustering of the garments. In this chapter, we experimentally analyse the performance of these processes and provide some ideas on how to improve them.

The experimentation has been conducted using a computer with the following specifications:

**CPU** Intel Core i7-3630QM 2.4GHz

**Graphics card** NVIDIA GeForce GT 640M with 2GB dedicated VRAM

**RAM** 8GB DDR3

**Primary HD** OCZ VERTEX 2 SATA II 2.5” SSD

**Secondary HD** Serie ATA 750GB

**OS** Windows 8 64-bit

## 6.1 Similarity matrix computation

Computing the similarity matrix of a set of garments requires: loading the garments images, computing the images masks and computing the similarity index for each of the observed attributes. The computation of the various similarity indexes is performed using image processing techniques and natural language analysis techniques (see section 5.5). Given the amount of garments to compare, the described processes require a lot

| | time (s) | % |
|---|---|---|
| load images and compute masks | 65.63 | 18.63 |
| compute histogram similarity | 114.18 | 32.41 |
| compute shape similarity | 47.58 | 13.50 |
| compute noise similarity | 109.89 | 31.19 |
| compute description similarity | 14.97 | 4.25 |

Table 6.1: Table showing the time spent in each of the steps of the similarity matrix computation and the percentage over the overall time.

of computational power. In what follows, we analyse the performance of the process of computing the similarity matrix of a set of garments.

Table 6.1 shows the time spent in each of the steps of the similarity matrix computation of a set of 468 garments. The first column shows the absolute time while the second column shows the percentage over the total time. The last column of the table shows the percentage of time spent on each of the steps over the total time.

As can be seen, the time spent computing the similarity matrix for a set of garments is mainly spent in applying the histogram similarity function and the noise similarity function. This is because these processes require the application of image processing techniques to the garments images and masks. Although the shape similarity function also requires the application of image processing techniques, the time spent for that function (13.50%) is nearly a third of the time spent computing the histogram similarity (32.41%) and the noise similarity (31.19%). This is due to the optimized implementation of the shape similarity function. When computing the similarity matrix slice corresponding to the shape similarity of a set of garments, the widths of each garment are only computed once. In contrast, when computing the slice corresponding to the noise similarity function, the garment edges image of a garment is computed each time it is compared with another garment. The same happens with the histogram similarity function, each time a garment is compared its histograms are extracted. To reduce the time spent computing the similarity matrix of a set of garments, the noise similarity function and histogram similarity function should avoid re-computing the data each time a garment is compared. That is, they should avoid recomputing the garment edge images and the garment image histograms array respectively.

Although there is still room for improvement in the computation of the similarity matrix, this task is very computational expensive. In what follows, we will analyse the time spent computing the similarity matrix of sets of garments with different number of garments. From the observations, we will determine the function that describes that value.

| #garm. | 145 | 179 | 288 | 327 | 486 | 716 | 904 | 1010 |
|---|---|---|---|---|---|---|---|---|
| #comp. | 10585 | 16110 | 41616 | 53628 | 118341 | 256686 | 409060 | 510555 |
| time | 28.87 | 44.22 | 113.07 | 162.22 | 350.76 | 848.91 | 1322.00 | 1663.66 |
| tpc | 0.0027 | 0.0027 | 0.0027 | 0.0030 | 0.0029 | 0.0033 | 0.0032 | 0.0032 |

Table 6.2: Table showing the number of garments comparisons made, the time spent and the average time spent per comparison when computing the similarity matrix of various sets of garments.
"#garm." stands for "number of garments".
"#comp." stands for "number of comparisons".
"tpc" stands for "time per comparison"

| num. garments | 145 | 179 | 288 | 327 | 486 | 716 | 904 | 1010 |
|---|---|---|---|---|---|---|---|---|
| observed time | 28.87 | 44.22 | 113.07 | 162.22 | 350.76 | 848.91 | 1322.00 | 1663.67 |
| expected time | 31.72 | 48.28 | 124.72 | 160.72 | 354.66 | 769.66 | 1225.91 | 1530.08 |

Table 6.3: Table showing the observed time spent computing the similarity matrix of different sets of garments, and the expected time.

Table 6.2 shows the number of garments comparisons made, the time spent and the average time spent per comparison when computing the similarity matrices of various sets of garments. The row *num. comparisons* shows the number of comparisons made during the computation of the similarity matrix. This value corresponds to the number of elements in the lower triangular matrix of the similarity matrix. This is true because, when computing the similarity matrix values, only the lower triangular matrix is computed. This simplification can be done because the similarity function used is commutative ($similarity(i, j) = similarity(j, i)$). The number of comparisons made is given by the formula $\frac{n^2-n}{2} + n$, where $n$ is the number of garments. The row *time* shows the observation of the time spent computing the the similarity matrix. This value is the mean of four observations. The row *mean time per comparison* shows the mean of the time spent doing a garment comparison ($\frac{time}{num.\ comparisons}$).

Table 6.3 shows the comparison between the observation of the time spent computing the similarity matrices of various sets of garments, and the expected time. The expected time spent computing the similarity matrix of a garment set is:

$$meanTimePerComparison \times (\frac{n^2 - n}{2} + n)$$

where $meanTimePerComparison$ (0.0029969022) is the mean of the values from the row *mean time per comparison* from the table 6.2 and $n$ is the number of garments in the set.

Figure 6.1 shows the plot of the data from the table 6.3. As can be seen, there is a deviation between the observed time and the expected time (the mean deviation is 6%). Despite that, the expected time is an informative measure of the time that will take to compute a similarity matrix.

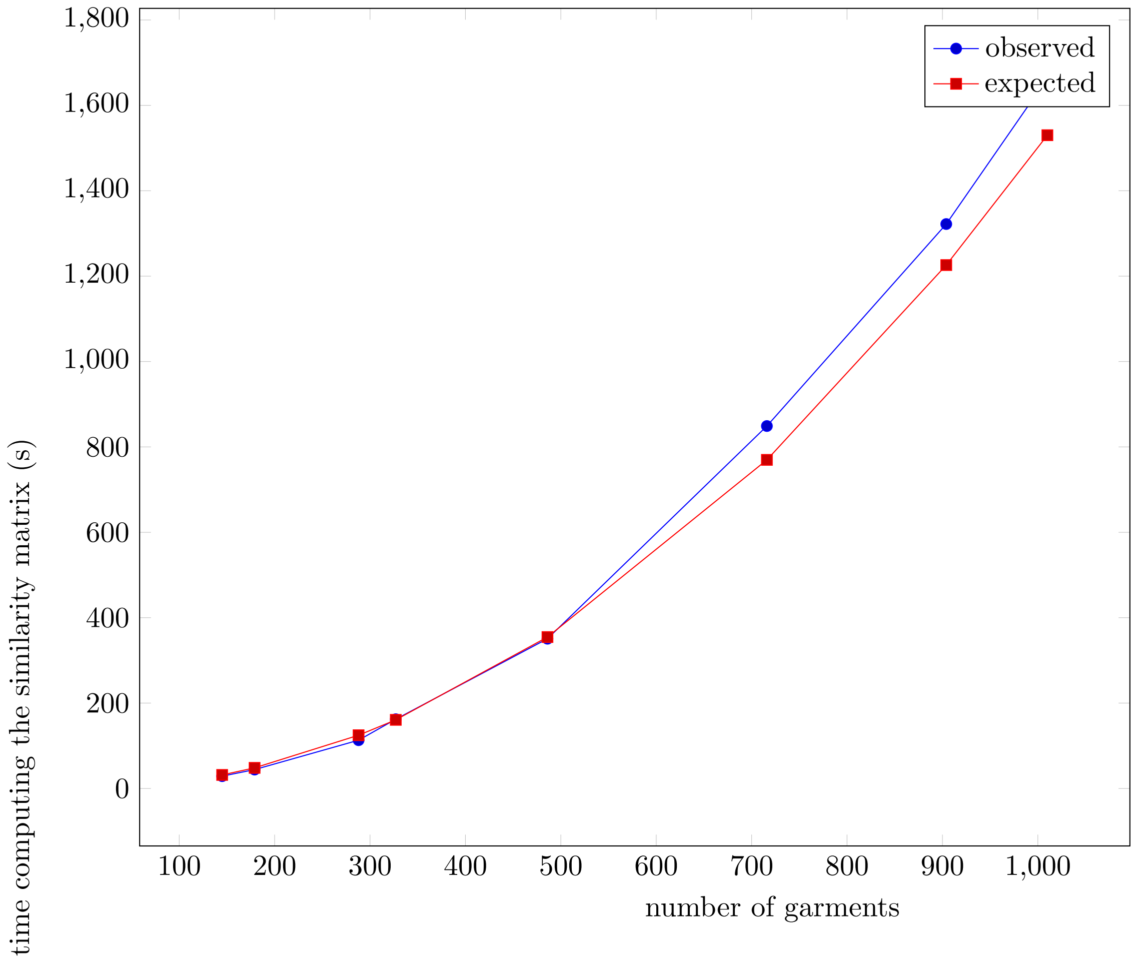


Figure 6.1: Garments clustering observed time and estimated time

## 6.2 Clustering

The second most computational expensive task performed by the recommendation system is the clustering of the garments. Once the similarity matrix of a set of garments has been computed, a spectral clustering algorithm is applied to that similarity matrix to compute the clusters. Table 6.4 shows the relation between the number of garments of a set and the time spent computing its clusters. The values from the column *clustering time* are the mean of four observations of the time spent clustering the garments. The $alpha-star$ parameter of the clustering algorithm was set to 0.999. Column *mean clustering time per garment* is the result of applying $\frac{clustering\ time}{garments}$. The correlation value between the number of garments and the clustering time is 0.906.

| garments | clustering time (s) | mean clustering time per garment |
|---|---|---|
| 486 | 4 | 0.0082 |
| 716 | 19 | 0.0265 |
| 757 | 24 | 0.0317 |
| 798 | 27 | 0.0338 |
| 804 | 27 | 0.0335 |
| 901 | 55 | 0.0610 |
| 904 | 58 | 0.0641 |
| 1010 | 68 | 0.0662 |
| 1099 | 113 | 0.1028 |
| 1192 | 160 | 0.1342 |
| 1299 | 150 | 0.1154 |
| 1319 | 155 | 0.1175 |
| 1428 | 200 | 0.1400 |
| 1506 | 240 | 0.1593 |
| 1622 | 345 | 0.2127 |
| 1668 | 409 | 0.2452 |
| 1907 | 706 | 0.3702 |

Table 6.4: Table showing the observations of the time spent clustering various sets of garments

Figure 6.2 shows the comparison between the clustering time observed and an the expected time. The expected value is given by the formula $expected(x) = 1.0 \times 10^{-9} \times x^{3.608}$. This formula has been obtained using an exponential regression technique. Although there is a deviation between the expected value and the observed value, we can conclude that it is a good approximation.

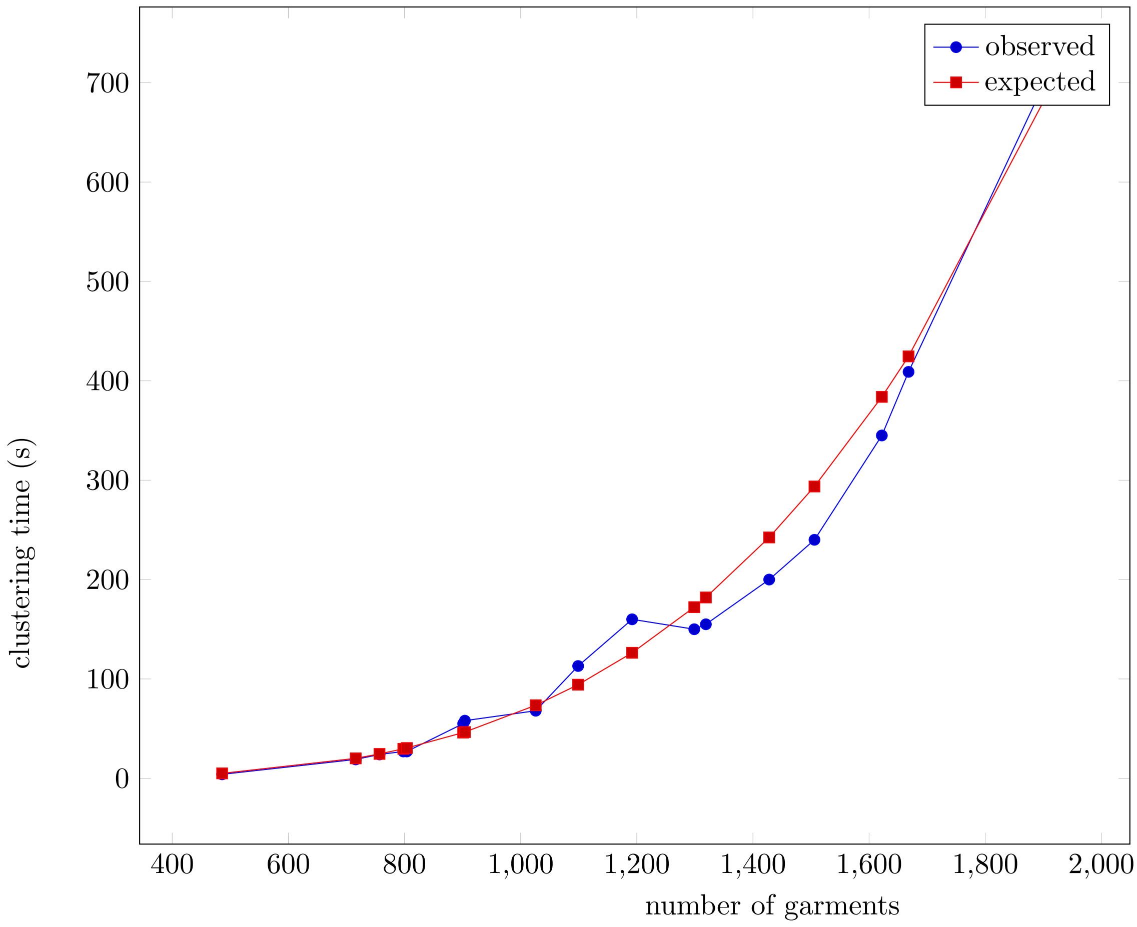


FIGURE 6.2: Garments clustering observed time and expected time

# Chapter 7

# Installation and execution

In this chapter, first, the installation process of the recommendation system is described. There, the steps that need to be taken to get the system working are explained. After that, the most relevant tasks that can be accomplished with the recommendation system are described. For each of these tasks, the steps needed to accomplish them are explained.

## 7.1 Installation

To install the recommendation system, the following steps need to be taken:

Install Java Standard Edition (SE) 1.7.X. The software and all the related documentation can be found at `http://www.oracle.com/`. When installing it, the version appropriated for the operating system (32-bits or 64-bits) needs to be chosen.

Download the source code an the initial data from the repository[1]. Place them in any folder of the file system with world read permissions. Grant full read and write permissions to the data folder and any of its contents.

Install MySQL 5.5 Community Edition. The software and all the related documentation can be found at `http://dev.mysql.com/`. Execute the script `CREATE.sql` found in the package `Core`. This script will create a database called `get_dressed_test` with all the necessary tables.

Install OpenCV 2.4.3. The software and all the related documentation can be found at `http://opencv.org/`. Instructions on how to install it in various platforms can be found at `http://opencv.org/quickstart.html`.

[1]`https://github.com/rsprat/outfit-recommendation-system.git`

Install JavaCV 0.3. The software and all the related documentation can be found at https://code.google.com/p/javacv/.

Install Eclipse Juno. The software and all the related documentation can be found at http://www.eclipse.org/.

Install Python 2.7.X. The software and all its related documentation can be found at http://www.python.org/.

Install LibXML2. The software and all its related documentation can be found at http://www.xmlsoft.org/.

Install Hibernate 4.1. The software and all its related documentation can be found at http://www.hibernate.org/.

Open the project from Eclipse. Under `project properties/Java build path/Libraries`, configure the paths to the libraries.

Modify the Java class `Paths` according to the paths to the source code and initial data.

Modify the file `hibernate.cfg.xml` found in the package `core` and set the appropriate parameters for the database connection.

To execute the Java code, write the following in the `VMarguments` text box under `Run/Run configuration/Arguments`:

```
-Djava.library.path="C:/Program Files/javacv/javacv-src/src/main/java/com/googlecode/javacv/cpp"
-Dlog4j.configuration=file:/${workspace_loc:outfitRecommendationSystem}/logconfig.lcf
-Xmx1000m
```

The argument $-Djava.library.path$ should point to the JavaCV installation folder.

## 7.2 Execution

In this section, the most relevant tasks that can be accomplished with the recommendation system are described. For each of these tasks, the steps needed to accomplish them are explained.

### Gather members initial data

To obtain a initial set of data relative to the fashion community members, we execute the Python script `getmembers.py` located in the folder \\`informationGathering`\\`dataRetrival`. This script starts gathering the data of a member (the one with id 349251) and keeps on gathering data of members he follows. For each gathered member, the members he follows are also gathered and so forth. All the captured data is stored in the files `member_data.csv` and `member_summary.csv` in the folder `\informationGathering\dataRetrival \captured`. Once the desired amount of data is captured, the script can be stopped. At any time, the script can be executed again and will continue gathering members data staring with the data that has not been already captured.

### Gathering outfits initial data

Once the script `getmembers.py` has been executed and data about the members has been gathered, the data of the outfits created by those members can be gathered. The Python script `getoutfits.py`, located in the folder `/informationGathering/ dataRetrival`, reads the data of the members gathered by the script `getusers.py`. For each member, traverses the list of outfits created by him, and for each outfit gathers its data. The data of the outfits is stored int the file `outfits.csv` located in the folder `/informationGathering/dataRetrival/captured`. The script can be stopped at any time. When re-executed, it will continue gathering the outfits that are not already in the outfits file.

### Gather garments initial data

Once the script `getoutfits.py` has been executed and data about the outfits has been gathered, the data of the garments that compose those outfits can be gathered. The Python script `getgarments.py`, located in the folder `/informationGathering/ dataRetrival`, reads the data of the outfits gathered by the script `getoutfits.py`. For each outfit, traverses the list of garments that compose it and gathers its data. For the gathered outfits, the data is stored in the file `garments.csv` located in the folder `/informationGathering/dataRetrival/captured`. The images of those outfits are stored in the folder `/informationGathering/dataRetrival/captured/images`. For the outfits that do not have a category or a textual description, its ids are stored in the file `nogamrnets.csv` located in the folder `/informationGathering/dataRetrival/ captured`.

### Import garments data

Once the data of the garments has been gathered from Polyvore and stored in CSV files, this data can now be stored in the database. To import the data from the CSV files to the database, we use the class `FillGarmentsFromFile` found in the package `rob.ors.informationGathering.filler`. Executing the main method of this class, all garments found in the file `garments.csv` are inserted in the database, excluding those found in the file `nogamrnets.csv`.

### Import outfits data

Once the data of the outfits has been gathered from Polyvore and stored in CSV files, this data can now be stored in the database. To import the data from the CSV files to the database,we use the class `FillOutfitsFromFile` found in the package `rob.ors.informationGathering.filler`. Executing the main method of this class, all outfits found in the file `outfits.csv` are inserted in the database.

### Import users data

Once the data of the members has been gathered from Polyvore and stored in CSV files, this data can now be stored in the database. To import the data from the CSV files to the database, we use the class `FillMembersFromFile` found in the package `rob.ors.informationGathering.filler`. Executing the main method of this class, all members found in the file `members.csv` are inserted in the database.

### Find not desired elements

Even though when gathering the garments, those that did not have an appropriate category or a description where discarded, some undesired elements might still exist. To remove these elements from the recommendation system,a visual tool has been developed (figure 7.1). The method `showSuspiciousGarmentsPanel()` from the class `View` found in the package `rob.ors.informationVisualitzation` shows a visual tool for detecting not desired elements. This tool analyses all the garment images and shows these that matches a defined criteria (see section 5.8). These elements can be deleted by clicking their image.

### Create abstract garments and abstract outfits

The generalization of the data involves the computation of the garments similarity matrices, the clustering of these garments (creating the abstract garments) and the creation of the abstract outfits.

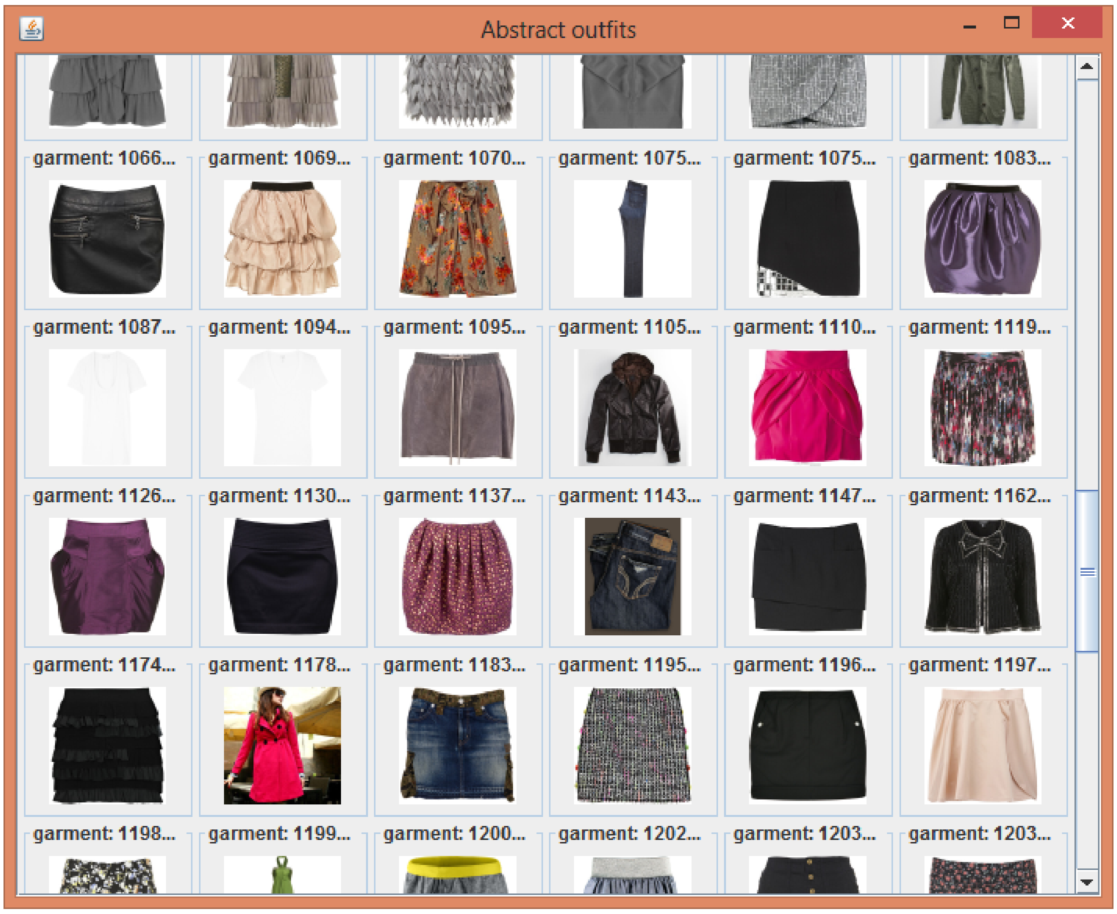


FIGURE 7.1: Visual tool for detecting non desired elements

The method `createSimilarityMatricesForAllCategories` from the class `Cluster` found in the package `rob.ors.garmentsclustering.clustering` computes the 3D similarity matrix of each category. After the execution of this method, the folder `/informationGathering/dataRetrival/captured/output` will contain a list of files called `sim_i.txt` where *i* is the id of a category. Each of those files will contain the different similarity matrices slices (one for each similarity function used) of the garments of a category.

After the similarity matrices for all the categories have been computed, the clustering algorithm can be applied to create the abstract outfits. The method `createAbstractGarmentsForAllCategories` from the class `Cluster` found in the package `rob.ors.garmentsclustering.clustering` creates the abstract outfits of each category. For each category, the 3D similarity matrix file found in the folder `/informationGathering/dataRetrival/captured/output` is read. That 3D similairty matrix is reduced to a 2D similarity matrix using the follwing weights: 0.5 for the colour similarity, 0.3 for the shape similarity, 0.2 for the noise similarity and 0.0 for the description similarity. Once the 2D similarity matrix is computed, the garments are clustered and the abstract clusters are stored in the database.

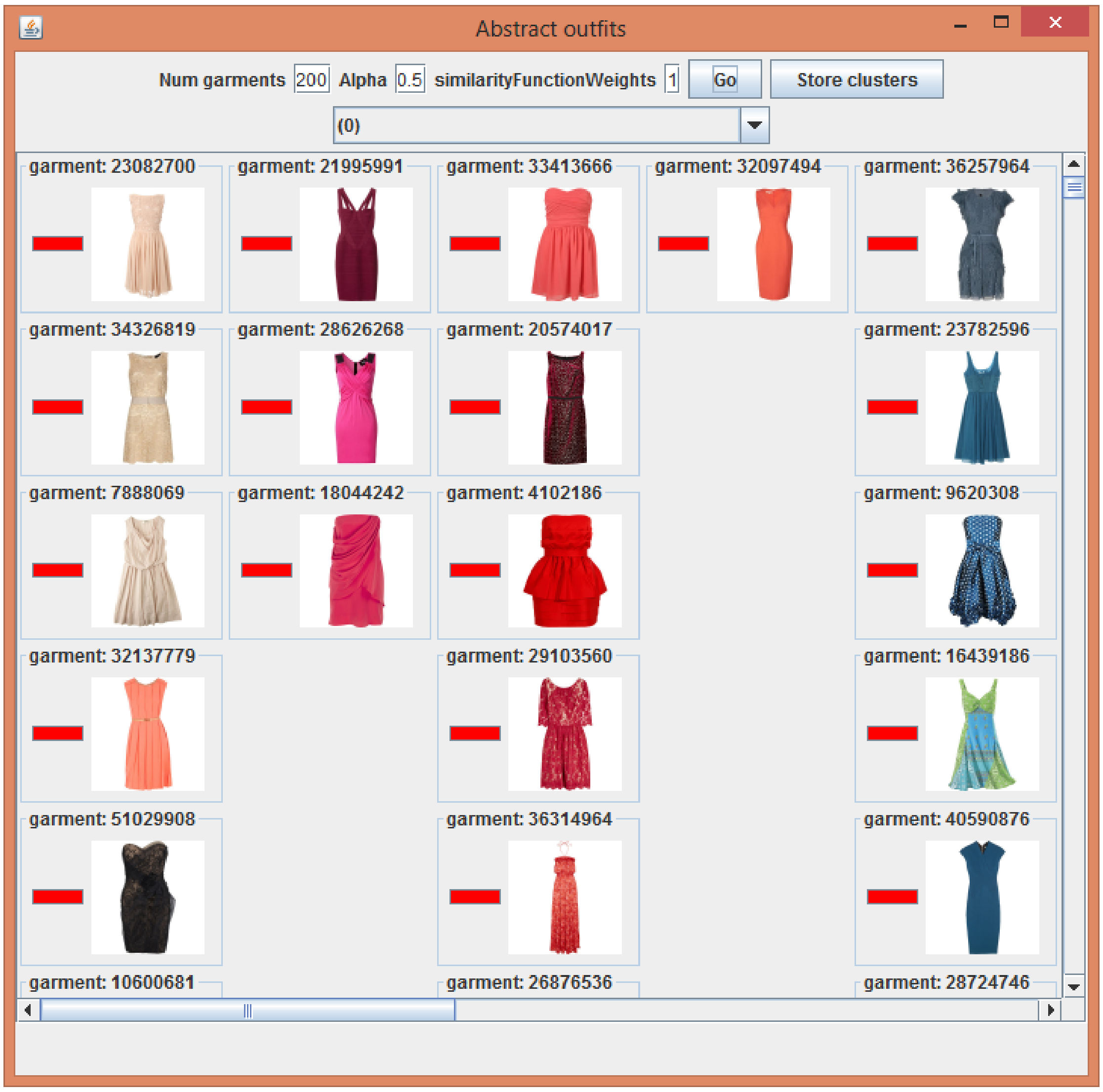


FIGURE 7.2: Visual tools for clustering the garments

The garments can also be clustered using a visual tool created for this task. The method`showClustersPanel(boolean readFiles)` from the class `View` found in the package `rob.ors.informationVisualitzation` shows a visual tool for clustering the garments. The parameter `readFiles` indicates if the similarity matrices are to be read from the folder `/informationGathering/dataRetrival/captured/output` or are to be computed on the fly.

Figure 7.2 shows a screen-shot of the visual tool for clustering the garments. At the top of the window, different parameters can be modified to affect the clustering results. Those parameters are:

**Num garments** the number of garments to be clustered.

**Alpha** the alpha-star value for the clustering algorithm. This value, ranging from 0.01 to 1, affects the deepness of the cuts performed by the algorithm, and thus, the number of resulting clusters.

**similarityFunctionWeight** the different weight values used to aggregate the 3D similarity matrix into a 2D similarity matrix. Those values, raging from 0 to 1, are separated by semicolons.

The button "Go" applies the changes made to the recently described parameters. The drop down box allows the user to choose the category to cluster. Finally, the button "Store clusters" stores the current clusters in the database. At the bottom of the window, the resulting clusters can be seen. The garments are divided in columns that represent clusters. If the user clicks the red button next to a garment, this will be removed from the category and stored in the no_garments table.

After creating the abstract garments, the abstract outfits can be created. The method `createAbstractOutfits` from the class `Cluster` found in the package `rob.ors.garmentsclustering.clustering` creates those abstract outfits based on the abstract garments in the system.

### See abstract outfits

After having created the abstract garments and abstract outfits, we can observe the later ones using the class `View`. This class has the method `showAbstractOutfitsPanel()` that shows a visual tool for analysing the abstract outfits.

Figure 7.3 shows a screen shot of the visual tool to analyse the abstract outfits. At the top of the window, there is a drop down menu for selecting the abstract outfit to visualize. Each element from this drop down has the form $abstractOutfitId\ \# : outfitCount\ items : garmentCount$ where $abstractOutfitId$ is the id of the abstract outfit, $outfitCount$ is the number of concrete outfits that conform this abstract outfit and $garmentCount$ is the number of abstract garments that this abstract outfit has. When an abstract outfit is selected from the drop down menu, the concrete outfits that conforms it are shown beneath. Each concrete outfit is displayed in a row. If the number of concrete outfits that conform the abstract outfit being analysed is greater than the number of concrete outfits that can be shown, the concrete outfits are paginated.

### Simulate a query to the system

After having performed the previous steps, we can now simulate the recommendation for a certain user about a garment. The class `View` has the method `showRecommendationPanel()`

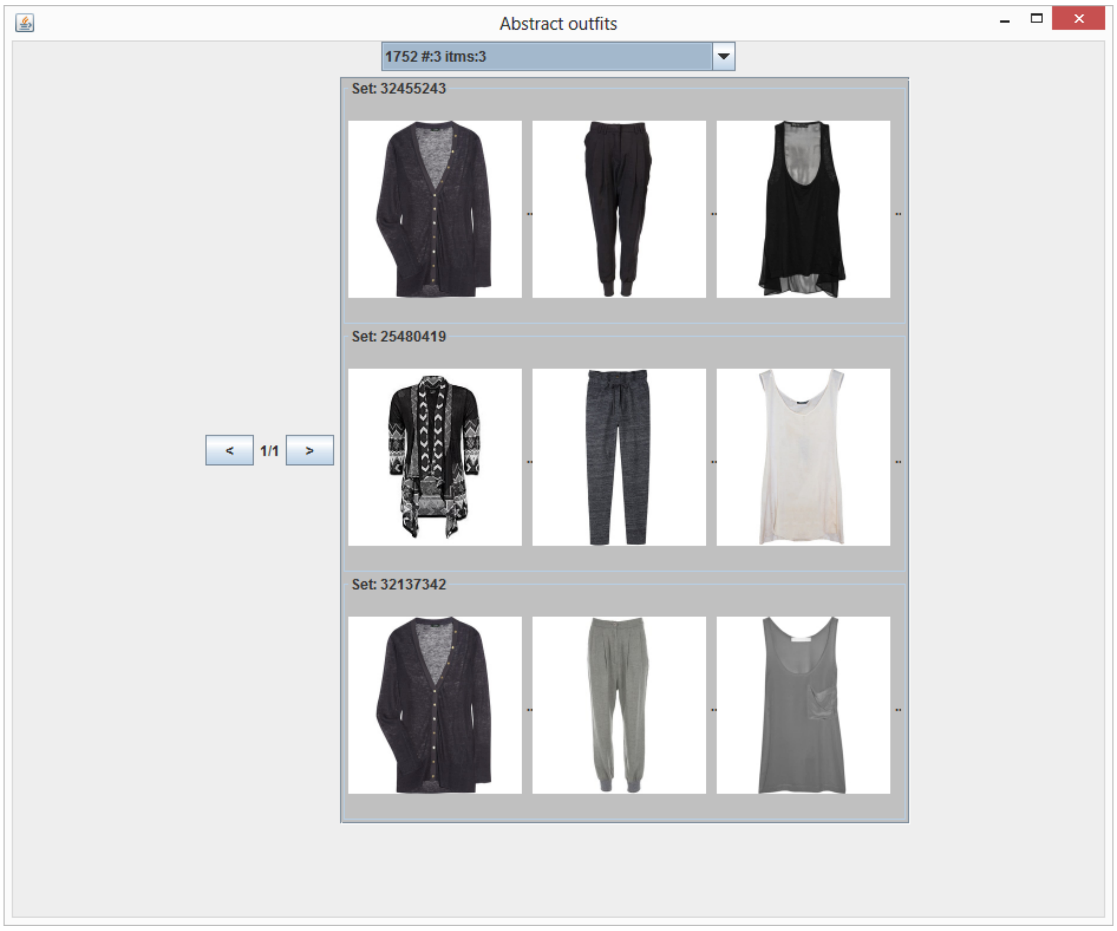
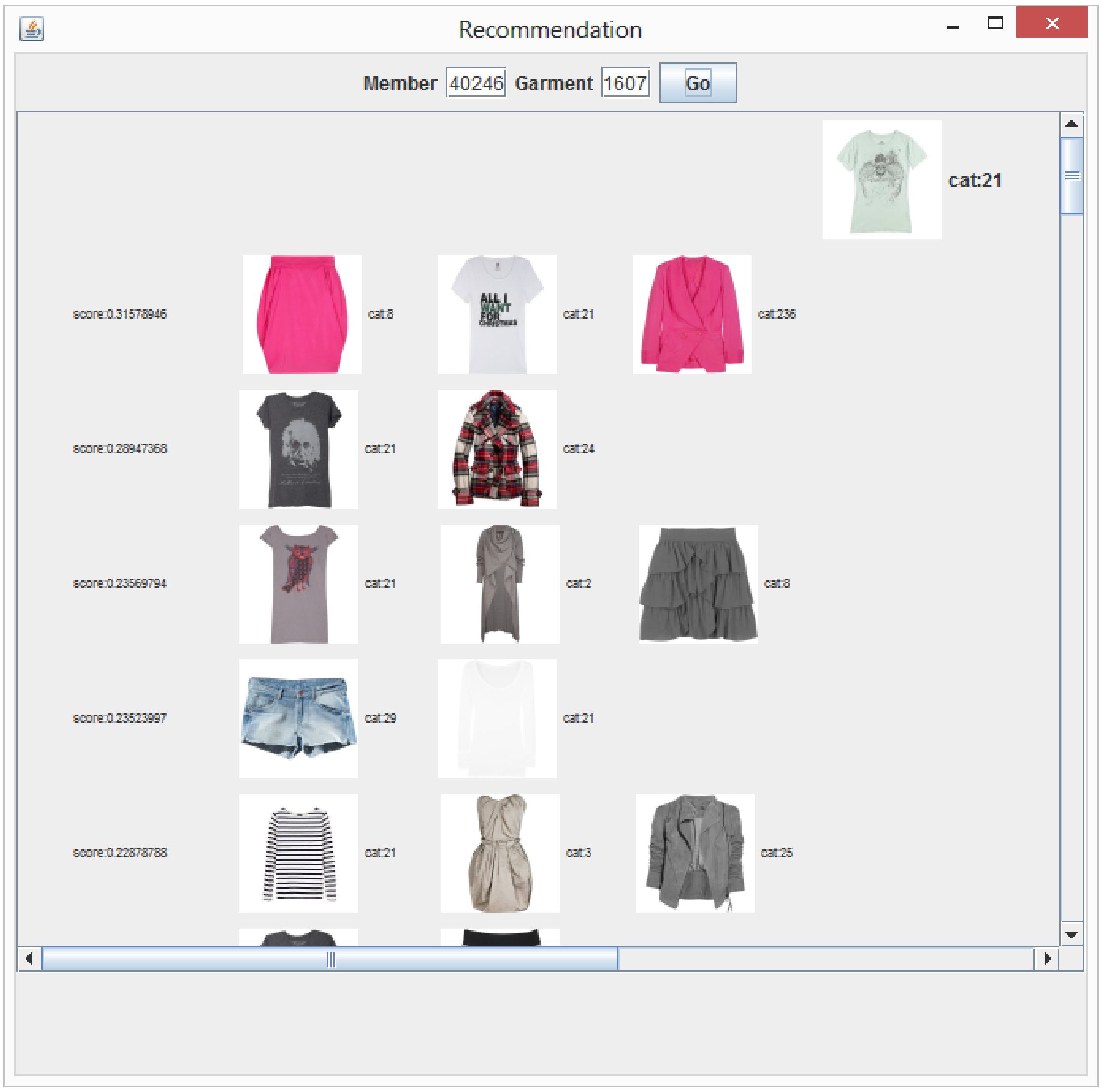


FIGURE 7.3: Visual tool for analysing the abstract outfits

that presents an user interface for simulating those answers. Figure 7.4 shows a screen shot of the visual tool that simulates answers to queries. At the top of the window, there are two input fields, one for the member id, and another for the garment id. After filling the fields and pressing the "Go" button, the recommendation system processes the query and presents the results. Beneath the input field, the image of the query garment are shown. Beneath that image, for each abstract outfit, a concrete outfit representing that abstract outfit, is shown. The concrete outfit shown is selected at random from the list of concrete outfits that compose the abstract outfit. Each concrete outfit is displayed on a single line with the score for the abstract outfit that is representing at the left.



```
Member 40246  Garment 1607

                                                        cat:21

score:0.31578946    cat:8     cat:21    cat:236
score:0.28947368    cat:21    cat:24
score:0.23569794    cat:21    cat:2     cat:8
score:0.23523997    cat:29    cat:21
score:0.22878788    cat:21    cat:3     cat:25
```

FIGURE 7.4: Visual tool for simulating answers to queries

# Chapter 8

# Project planing and costs

In this chapter, the process followed to develop the recommendation system is described. After that, the costs of the project are listed.

## 8.1 Project planing

After the problem to solve was clearly stated, the execution of the current project was divided into the following tasks:

**Search existing solutions** During the execution of this task, various on-line solutions related to the garments/outfits recommendations where explored. For each solution, its pros and cons where analized.

**Design the recommendation system** This task consisted in the design of the recommendation system structure. During this process, the modules that compose the solution where defined. For each module, its major functionalities where described as well as the module interaction with the rest of the modules.

**Select a data source** After having designed the recommendation system, many potential sources of data for the proposed recommendation system where studied. Among all the possible sources, Polyvore was chosen given the amount of data it could provide.

**Implement the core module** This task included the implementation of the domain model objects and the data persistence layer.

**Implement the information gathering module** During the execution of this task, the scripts that that gather data from Polyvore were developed. Also the the Java classes to integrate these scripts with the recommendation system where developed.

**Implement the garment similarity module** This task included the implementation of the processes for extracting attributes values from the garment images and textual descriptions. This task also included the implementation of the similarity functions used to compare two garments based on these attributes.

**Implement the garment clustering module** This task included the implementation of the processes for clustering the garments based on their similarities. Apart from that, this task also included the implementation of the process for constructing the abstract outfits.

**Implement the recommendation module** This task included the implementation of the process for making recommendations to answer user queries.

## 8.2 Costs

The costs of the project can be divided into: costs derived from the human resources needed, and costs derived from the software licences and hardware. In what follows, a listing of these costs is given:

### 8.2.1 Human resources

The human resources needed for the execution of the project are directly related with the time spent in the execution of the same.

Table 8.1, shows the time spent in each task and sub-task from the project plan. The times where precisely measured using an on-line time tracking tool called Toggl[29]. For each task, the required profile for performing it is shown. Two different profiles have been required for this project: analyst and developer. The analyst profile is focused on analysing the problem, researching existing solutions and designing the recommendation system. This profile has been assigned a hourly wage of 20€. The developer profile is focused on the implementation of the recommendation system designed by the analyst. The developer profile has been assigned a hourly wage of 10€.

As can be seen, the implementation of the garment similarity module has been the most time consuming task. This is because a lot of effort has been putt in deciding the appropriate attributes that should be observed for computing the similarity between garments. During the implementation of the garment similarity module, the computation of many attributes values form the garment image and garment textual description was explored. Finally, from all the explored solutions, it was decided to use the ones described in Chapter 4.4.

| task | sub-task | hours | required profile | cost (€) |
|---|---|---|---|---|
| Search existing solutions | | 17 | analyst | 340 |
| Design the recommendation system | | 40 | analyst | 800 |
| Select a data source | | 5 | analyst | 100 |
| Implement the core module | | 57 | | 570 |
| | Implement the Java model classes | 25 | developer | 250 |
| | Implement the persistence layer | 32 | developer | 320 |
| Implement the information gathering module | | 57 | | 570 |
| | Implement the data gathering scripts | 40 | developer | 400 |
| | Implement the Java integration of data gathering scripts | 17 | developer | 170 |
| Implement the garments similarity module | | 80 | | 800 |
| | Implement the image features extraction | 60 | developer | 600 |
| | Implement the textual description analysis | 20 | developer | 200 |
| Implement the garments clustering module | | 32 | developer | 320 |
| Implement the visual tools | | 37 | developer | 370 |
| Total | | 325 | | 3870 |

Table 8.1: Listing of the time spent performing the tasks and sub-tasks that composed the project. The cost of each task is also listed.

### 8.2.2 Software and hardware resources

The execution of this project has been done using mostly open source software and tools (for a list of the technologies used see section 5.1). The only privative software used has been a copy of Windows 8.

The only hardware requirements for this project has been the personal computer used.

# Chapter 9

# Conclusions

In this chapter, a summary of the accomplished work is given and the biggest problems faced during the process are described. Finally, some ideas for future work are outlined.

In this project, we have explored the application of recommendation systems to the world of fashion. We have proposed a system capable of making recommendations on how to style a certain garment for a particular user. That is, our recommendation system is capable of, given a garment and a user, create an outfit that matches the user's taste and includes the query garment.

The recommendations given by our system are based in real outfits created by a community of fashion enthusiasts. The recommendation process starts by searching all the abstract outfits that contain the abstraction of the query garment. Then, the system gives a score to those abstract outfits, based on how likely they are to fit the taste of the user, its closest community members or the whole community. Finally, one of those abstract outfits is selected and translated to a concrete outfit. This concrete outfit is then sent as a response to the query.

We have introduced the concept of abstract garments and abstract outfits to overcome the sparsity of the original data. We defined an abstract garment as a cluster of garments. These clusters have been created applying an spectral clustering technique using a custom similarity function. The similarity function used computes an overall similarity index. This overall similarity index is the weighted aggregation of different similarity indexes with respect to various observed attributes (colour, shape, pattern, etc.). The values for those attributes are computed using image processing techniques and natural language analysis techniques.

The most challenging problem faced during the execution of this project has been to find a suitable methodology to reduce the data sparsity through clustering the garments. A big effort has been put on identifying significant attributes for comparing the garments and designing the procedures to compute the values for those attributes.

Another problem faced during the execution of this project has been the access to the data. Polyvore does not expose a public API, and our request for having access to their database has been unanswered. Given that, have developed a series of scripts to automatically gather the data from the HTML sent by the web site. Although have gathered a subset of the data to work with it, we haven't been able to access the whole database.

As a result of the work done, we have published an open-source recommendation system that given a query from an user about a certain garment, is capable of creating an outfit that fits the user's taste.

## 9.1 Future work

The future lines to extend this project might be:

**Improve data source:** find an on-line fashion community that is willing to collaborate in the project, allowing access to its database.

**Improve garment clustering:** one of the key aspects of the recommendation system is the clustering of the garments. Although we have developed a clustering process that, from our point of view, is on the right track for providing meaningful results, more work needs to be done in this area. If the time constraints would not have been so strict, we would have, for example, explored the possibility of improving the detection of patterns in the garments. We are sure that improvements in the clustering of the garments would carry improvements in the overall results.

**Improve recommendation process:** improvements can be made to the recommendation process. For instance, exploring different selection algorithms rather than the roulette algorithm for the selection of a concrete outfit from an abstract outfit. Another possible improvement would be to select the score function weights ($w_u, w_f$ and $w_c$) depending on how much data we have for a certain user. For instance, a user that actively participates in the fashion community by creating and liking outfits should have a bigger value for the $w_u$ than a user with less participation. This way the user's taste would be more prevalent in recommendations for users about whom we have more information on his taste.

**Members clustering:** analyse the social network and find clusters of members with similar taste. Right now, the recommendation system is highly dependent on the data about the outfits the user has created or liked. Making it necessary for the user to be active in the social network. The system needs this interaction to gather data about the user taste. To make the system more exploitable, it would be interesting to eliminate the necessity for the user to participate in the community. To this end, one of the possible solutions would be to cluster the community members based on their taste. Extracting the most characteristic outfits created by those clusters, we could create a user classification algorithm based on a decision tree. Then, when a user from whom we do not have any previous data, wants to make a query to the system, he would first be presented with a series of outfits. With his assessment on those outfits, the decision tree would be able to assign that user to a cluster of existing members. This way, we would automatically have some data about the taste of the user.

**Apply the recommendation system:** the proposed recommendation system, or a slight modification of it, can be used in many real applications. In this project we have focused on giving advice on how to style a certain garment. Another possible application would be to recommend the purchase of new garments. Having a digital version of the user wardrobe, it could be possible to give advice on which new garments are more suitable for the user based on how many new outfits the user can create with those garment. This tool could be implemented into end user applications or even used as a recommendation system for e-commerce sites.

# Appendix A

# Garments categories

Figure A.1 shows the list of garment categories the recommendation system supports. The structure of the categories as well as their name and ids are the same as the ones used in Polyvore.

- **Fashion** 1
  - **Clothing** 2
    - **Dress** 3
      - **Day** 4
      - **Cocktail** 5
      - **Grown** 6
    - **Skirt** 7
      - **Mini** 8
      - **Knee length** 9
      - **Long** 10
    - **Tops** 11
      - **Blouse** 17
      - **Cardigan** 18
      - **Sweater** 19
      - **Sweatshirts & hoodies** 20
        - **Sweatshirts** 4495
        - **Hoodies** 4496
      - **Tank** 104
      - **T-shirt** 21
      - **Tunic** 15
    - **Outwear** 23
      - **Coat** 24
      - **Jacket** 25
        - **Blazers** 236
      - **Vest** 26
    - **Jeans** 27
      - **Bootcut** 238
      - **Boyfriend** 240
      - **Skinny** 237
      - **Straight** 236
      - **Wide** 239
    - **Pants** 28
      - **Cappry & Croped** 332
      - **Legins** 241
    - **Short** 29
    - **Jumpsuit & Romper** 242
      - **Jumpsuit** 243
      - **Romper** 241

Figure A.1: Full list of garments categories from the virtual wardrobe, the number aside of every category is the category identification number.

Appendix B

# Contents of the file `attribute.xml`

Figure B.1 shows the content of the file `attributes.xml`. For further reference on the structure of this file, please read the description of the the implementation of the garment similarity package (section 5.5).

```
<?xml version="1.0" encoding="UTF-8"?>
<root>

    <attribute name="neck" id="1" defaultId="25">
        <values>
        <value name="v" id="1">
         <synonym name="'v'"/>
        </value>
    <value name="drapped" id="2"/>
<value name="halter" id="3"/>
<value name="square" id="4"/>
<value name="crew" id="5"/>
<value name="scoop" id="6"/>
<value name="boat" id="7">
    <synonym name="bateau"/>
</value>
<value name="turtle" id="8">
    <synonym name="polo"/>
    <synonym name="turtleneck"/>
</value>
<value name="off-shoulder" id="9">
    <synonym name="off-the-shoulder"/>
</value>
<value name="sweetheart" id="10"/>
<value name="keyhole" id="11"/>
<value name="cowl" id="12"/>
<value name="round" id="13"/>
<value name="plunging" id="14"/>
<value name="jewel" id="15"/>
<value name="one-shoulder" id="16" >
    <synonym name="one shulder"/>
</value>
<value name="peter pan" id="17">
    <synonym name="peter-pan"/>
</value>
<value name="sailor" id="18"/>
<value name="mandarin" id="19"/>
<value name="roll" id="20"/>
<value name="cutout top" id="21"/>
<value name="funnel" id="22"/>
<value name="draped" id="23"/>
<value name="slash" id="23"/>
<value name="high" id="23"/>
<value name="twisted neckline" id="24"/>
<value name="UNKNOWN" id="25"/>

</values>
<automatic-synonyms>
    <append term=" neck"/>
    <append term=" neckline"/>
    <append term=" necklined"/>
    <append term=" collar"/>
</automatic-synonyms>
    </attribute>
```

# Bibliography

[1] Oracle Corporation and/or its affiliates. *MySQL :: MySQL Workbench 6.0.* [Online; accessed 28-August-2013]. 2013. URL: http://www.mysql.com/products/workbench/.

[2] Inc. or its affiliates Amazon.com. *Amazon.com: Online Shopping for Electronics, Apparel, Computers, Books, DVDs & more.* [Online; accessed 28-August-2013]. 2013. URL: http://www.amazon.com/.

[3] Samuel Audet. *JavaCV.* [Online; accessed 28-August-2013]. URL: https://code.google.com/p/javacv/.

[4] Pascal Brachet. *Texmaker (free cross-platform latex editor).* [Online; accessed 28-August-2013]. 2013. URL: http://www.xm1math.net/texmaker/.

[5] John Canny. "A Computational Approach to Edge Detection". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* PAMI-8.6 (1986), pp. 679–698. ISSN: 0162-8828. DOI: 10.1109/TPAMI.1986.4767851.

[6] CERN. *Colt.* [Online; accessed 28-August-2013]. 2004. URL: http://acs.lbl.gov/software/colt/.

[7] JBoss Community. *Hibernate - JBoss Community.* [Online; accessed 28-August-2013]. URL: http://www.hibernate.org/.

[8] Luigi Dragone. *Spectral Clusterer for WEKA.* [Online; accessed 28-August-2013]. URL: http://www.luigidragone.com/software/spectral-clusterer-for-weka/.

[9] Inc. Fashism. *Where the best dressed people rule!* [Online; accessed 28-August-2013]. 2013. URL: http://fashism.com/.

[10] The Eclipse Foundation. *Eclipse - The Eclipse Foundation open source community website.* [Online; accessed 28-August-2013]. 2013. URL: http://www.eclipse.org/.

[11] Inc. Free Software Foundation. *Licenses - GNU Project - Free Software Foundation*. [Online; accessed 28-August-2013]. 2013. URL: http://www.gnu.org/licenses/licenses.en.html.

[12] Fumiko Harada, Yukiko Okamoto, and Hiromitsu Shimakawa. "Outfit recommendation with consideration of user policy and preference on layered combination of garments". In: *International Journal of Advanced Computer Science* 2.2 (2012).

[13] Don Ho. *Notepad ++ Home*. [Online; accessed 28-August-2013]. 2011. URL: http://notepad-plus-plus.org/.

[14] GoTryItOn Inc. *Go Try It On*. [Online; accessed 10-August-2013]. 2013. URL: http://www.gotryiton.com/.

[15] Sangoh Jeong. *Histogram-Based Color Image Retrieval*. [Online; accessed 28-August-2013]. 2001. URL: http://scien.stanford.edu/pages/labsite/2002/psych221/projects/02/sojeong/.

[16] Dave Kuhlman. *Dave's Page*. [Online; accessed 28-August-2013]. 2012. URL: http://www.rexx.com/~dkuhlman/.

[17] Mycrosoft. *Microsoft Windows*. [Online; accessed 28-August-2013]. 2013. URL: http://windows.microsoft.com/en-gb/windows/home.

[18] Inc. Netflix. *Netflix*. [Online; accessed 28-August-2013]. 2013. URL: http://www.netflix.com.

[19] Inc. Netflix. *Netflix Prize: Home*. [Online; accessed 28-August-2013]. 2013. URL: http://www.netflixprize.com/.

[20] Oracle. *Java*. June 2009. URL: http://www.oracle.com/technetwork/java/index.html.

[21] Polyvore. *Polyvore*. [Online; accessed 28-August-2013]. 2013. URL: http://www.polyvore.com/.

[22] Python. *Python Programming Language  Official Website*. [Online; accessed 28-August-2013]. URL: http://www.python.org/.

[23] J.Ben Schafer et al. "Collaborative Filtering Recommender Systems". In: *The Adaptive Web*. Ed. by Peter Brusilovsky, Alfred Kobsa, and Wolfgang Nejdl. Vol. 4321. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 291–324. ISBN: 978-3-540-72078-2. DOI: 10.1007/978-3-540-72079-9_9. URL: http://dx.doi.org/10.1007/978-3-540-72079-9_9.

[24] Teruji Sekozawa. "One to one recommendation system for apparel online shopping". In: *WTOS* 9.1 (Jan. 2010), pp. 94–103. ISSN: 1109-2777. URL: http://dl.acm.org/citation.cfm?id=1853771.1853781.

[25] Edward Shen, Henry Lieberman, and Francis Lam. "What am I gonna wear?: scenario-oriented recommendation". In: *Proceedings of the 12th international conference on Intelligent user interfaces.* ACM. 2007, pp. 365–368.

[26] opencv dev team. *Histogram Comparison OpenCV 2.4.6.0 documentation.* [Online; accessed 28-August-2013]. 2013. URL: `http://docs.opencv.org/doc/tutorials/imgproc/histograms/histogram_comparison/histogram_comparison.html`.

[27] OpenCV Developers Team. *About — OpenCV.* [Online; accessed 28-August-2013]. URL: `http://opencv.org/about.html`.

[28] OpenCV Developers Team. *OpenCV — OpenCV.* [Online; accessed 28-August-2013]. URL: `http://opencv.org`.

[29] Toggl. *Toggl - Insanely simple time tracking.* [Online; accessed 28-August-2013]. URL: `https://www.toggl.com/`.

[30] Daniel Veillard. *The XML C parser and toolkit of Gnome.* [Online; accessed 28-August-2013]. 2004. URL: `http://acs.lbl.gov/software/colt/`.

[31] Machine Learning Group at the University of Waikato. *Weka 3 - Data Mining with Open Source Machine Learning Software in Java.* [Online; accessed 28-August-2013]. URL: `http://www.cs.waikato.ac.nz/ml/weka/`.

[32] Wikipedia. *Comma-separated values — Wikipedia, The Free Encyclopedia.* [Online; accessed 3-September-2013]. 2013. URL: `http://en.wikipedia.org/w/index.php?title=Comma-separated_values&oldid=570343716`.

[33] Wikipedia. *Hibernate (Java) — Wikipedia, The Free Encyclopedia.* [Online; accessed 28-August-2013]. 2013. URL: `http://en.wikipedia.org/w/index.php?title=Hibernate_(Java)&oldid=569071870`.

[34] Wikipedia. *Java (programming language) — Wikipedia, The Free Encyclopedia.* [Online; accessed 28-August-2013]. 2013. URL: `http://en.wikipedia.org/w/index.php?title=Java_(programming_language)&oldid=570196473`.

[35] Wikipedia. *MySQL — Wikipedia, The Free Encyclopedia.* [Online; accessed 29-August-2013]. 2013. URL: `http://en.wikipedia.org/w/index.php?title=MySQL&oldid=569693962`.

[36] Wikipedia. *Python (programming language) — Wikipedia, The Free Encyclopedia.* [Online; accessed 28-August-2013]. 2013. URL: `http://en.wikipedia.org/w/index.php?title=Python_(programming_language)&oldid=569966816`.

[37] Wikipedia. *Windows 8 — Wikipedia, The Free Encyclopedia.* [Online; accessed 28-August-2013]. 2013. URL: `http://en.wikipedia.org/w/index.php?title=Windows_8&oldid=570418265`.