FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FEUP

Content Blaster: the online show generator

Paulo Eduardo Gonçalves de Freitas Pereira

Report of Project

Master in Informatics and Computing Engineering

Supervisor: Ademar Aguiar (PhD.)

17^{th} July, 2009

Content Blaster: the online show generator

Paulo Eduardo Gonçalves de Freitas Pereira

Report of Project
Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: Ana Paula Cunha da Rocha (Auxiliar Professor)

External Examiner: Carlos Miguel Ferraz Baquero Moreno (Auxiliar Professor)
Internal Examiner: Ademar Manuel Teixeira de Aguiar (Auxiliar Professor)

17^{th} July, 2009

Abstract

As a consequence of the Web 2.0 phenomenon in recent years, people are no longer mere consumers of the web—they're also producers. If the amount of online information was already beginning to be a problem that motivated the creation of search engines, the surge of user-generated content made it clear that search engines aren't enough to filter through all the available information. The web is suffering from an information overload, and even our natural filtering skills are no match for it.

However, there's one technique which is very helpful to filter through the immense amount of information available today—recommendations. Everyone's friends, family and colleagues know what each other like and inform them about what may be of their interest.

Recommender systems automate this collaborative *word of mouth* behavior, providing recommendations to their users based on other like-minded individuals have enjoyed in the past. As these systems are usually web-based, people need to be sitting in front of their computers, which is a major shortcoming. However, people carry their mobile phones everywhere, and many enjoy the comfort of their sofas when watching TV, which may be connected to a set-top-box—but they don't have recommendations on those devices.

This report introduces Content Blaster, a project that seeks to fulfill what's missing in recommendation systems, providing a web interface for recommendations but also a programming interface, so applications can be created for any online device, extending the usefulness of recommendations to a broader audience.

It reviews the state of the art in recommendation systems and proposes a solution for ubiquitous recommendations of webpages. The solution takes advantage of both Slope One, a simple but accurate recommendation algorithm with good performance, and *filterbots*, automatized agents that rate webpages based on their tags.

Technologies used are then explained, and design and implementation details are also provided, with a special focus on the recommendation algorithms used and the usefulness of the Ruby on Rails framework during development. It then evaluations the proposed solution in terms of performance, and details the conclusions, limitations and benefits of the project. Finally, the report provides some insights on how the system could be improved in the future with enhanced content visualization, some additional new features and applications in other devices.

Resumo

Como consequência do fenómeno da Web 2.0 nos últimos anos, as pessoas já não são meros consumidores da *web*—são também produtores. Se a quantidade de informação disponível *online* era já um problema que levou à criação de motores de busca, a explosão de conteúdo criado por utilizadores tornou claro que os motores de pesquisa não são suficientemente capazes de filtrar toda esta informação. A *web* sofre de um problema de *excesso de informação*, e nem os nossa capacidade natural de filtrar as coisas é capaz de o enfrentar.

Não obstante, existe um método natural extremamente útil para filtrar informação: *recomendações*. Cada um sabe o que os seus amigos, família e colegas gostam, e informam-nos do que pensam ser do seu interesse.

Os *sistemas de recomendação* automatizam este processo natural, dando recomendações aos seus utilizadores baseado no que outros indivíduos com preferências semelhantes gostaram no passado. Sendo estes sistemas tradicionalmente baseados na web, obrigam a que os seus utilizadores estejam sentados em frente aos seus computadores, o que nem sempre é desejável. No entanto, as pessoas carregam consigo os seus telemóveis e muitas têm em casa uma *set-top-box* ligada à televisão. Mas não conseguem ter recomendações nesses aparelhos.

Este relatório apresenta o Content Blaster, um projecto que visa colmatar o que falha nos sistemas de recomendação, criando uma interface *web* que disponibiliza recomendações, mas também uma interface que permite que qualquer programador leve as recomendações até uma aplicação para qualquer dispositivo com uma ligação à Internet. Efectivamente, isto permite levar a utilidade das recomendações a uma maior audiência.

O relatório também revê o estado da arte dos sistemas de recomendação e propõe uma solução para recomendações ubíquas de páginas web. Esta solução tira partido do *Slope One*, um algoritmo simples e com boa *performance*, mas que oferece ao mesmo tempo recomendações de qualidade. Este algoritmo combina também a utilização de *filterbots*, agentes automatizados que votam nas páginas web baseando-se para isso nas suas *tags*.

Seguidamente são explicadas e detalhadas as tecnologias utilizadas, bem como o desenho e a implementação do sistema, com particular pormenor nos algoritmos utilizados e na utilidade prática da utilização da *framework* Ruby on Rails. É também avaliada a solução proposta em termos de *performance*, e apresentadas as conclusões, limitações e benefícios do projecto. Finalmente, são referidas algumas possibilidades de desenvolvimento futuro, como a melhoria na visualização dos conteúdos, funcionalidades adicionais e aplicações em outros dispositivos.

Acknowledgements

To everyone at Fraunhofer Portugal for making it an extraordinary place to work, specially to all the students and the IT guys for the fun.

To Filipe Abrantes and Ademar Aguiar for the patience needed to supervise an annoying and sometimes lazy student.

To the online communities for being so helpful and passionate, whose spend their time helping people so everyone can get better on what they do.

To Levi Figueira for revising this report and his questions about Rails that keep pushing me ahead.

To my colleagues back at the design classes, for teaching me other ways to look at things.

To everyone I worked, studied or partied during these years, specially to the VMT group for the exchange of knowledge, frustration and motivation—we did it!

To everyone who dressed in black like me for so many days and nights, specially to those I shared a spoon with, you shaped me more than you think.

To everyone at NIFEUP, for our discussions (even the pointless ones), all the mockery, the long nights spent working, the long nights spent *not* working, the weird stuff that happened sometimes, for helping me and letting me help them. Thank you all for being such awesome friends and teaching me so much.

To Joana, for her love, strength and support when things didn't look pretty at all.

To my family for their love, specially to my mother, who did everything beyond imaginable during these years so I could be here finishing my thesis.

"Simplicity is the ultimate sophistication"

— Leonardo DaVinci

Contents

List of Figures

LIST OF FIGURES

List of Tables

Definitions

3G Third generation of mobile phone communication standards.

Click-through rate The rate of people clicking on a particular item out of the total number who see the item.

Conversion rate The rate of clicks that result in a commissionable activity such as a sale or lead.

Netbook A small and economical laptop computer designe for basic actions such as text processing or wireless browsing.

RESTful The property of a system that complies to the REST principles.

Set-top box A device that enables the display of cable or satellite signal on a television.

Abbreviations

API Application Programming Interface

HTTP Hypertext Transport Protocol

JSON JavaScript Object Notation

MVC Model-View-Controller

ORM Object-Relational Mapping

REST Representational state transfer

RSS Really Simple Syndication

UGC User-Generated Content

URI Uniform Resource Indicator

URL Uniform Resource Locator

XML Extensible Markup Language

ABREVIATIONS

Chapter 1

Introduction

This section briefly presents the project's context, purpose and scope, while also detailing the report structure, providing an overview of each of the remaining chapters.

1.1 Context

In recent years, with the increasing popularity of Web 2.0 platforms and technologies such as blogging, social networks and media-sharing platforms, there has been a massive growth in content at our disposal. Internet users stopped being mere consumers and started creating content, taking advantage of the proliferation and ease of use of the new platforms and tools. As of today, a lot of *user-generated* content is watched and created every day [Cer07]. Take YouTube as an example, which doubled the video length getting uploaded per minute between 2007 and 2009 [Rya09] and now places third in the Alexa Traffic Rank, that tracks the world's most visited websites [Alea].

The World Wide Web changed its face completely, and half of the top ten websites in the Alexa Traffic Rank are now social networks, blogging platforms or media-sharing sites (see Appendix A). Platforms like these empower the production and publishing of *user-generated content*, the kind of content responsible for the enormous growth of the web in recent years. While the quantity and availability of content may be what makes the web great, it also poses a problem to users as they have to filter as much information as possible in order to find their way around this massive web. As [Cer07] put it,

> "(...) it seems likely that unless we learn to harness the energy unleashed by the Internet, we will soon be buried in an avalanche of information."

The information overload on the web is a long-standing issue that led to the creation of multiple *search engines*. Furthermore, the one which used a particular approach to ranking webpages was the one which stood above the competition—Google. They created and

patented the PageRank algorithm [Goo], which takes in account the number of inbound links of a page—to put simply, if many sites link to a particular page, it's because that's a good one [Seg07]. The hyperlink is perceived as the way of a webpage saying "this other webpage has relevant content", and by doing this on an huge scale, Google capitalized on the *opinion* of the entire web about itself. Using this approach, Google was doing something entirely different than just indexing webpages—they were entering the field of collective intelligence, defined as

> "(...) the combining of behavior, preferences, or ideas of a group of people to create novel insights." [Seg07]

Collective intelligence isn't something entirely new—everyone does it, when asking for *opinion* or advice to family, friends and coworkers. However, websites are capable of using the world wide web to collect *opinion* at a massive scale, and most importantly, gather valuable new information from it.

Others have followed behind Google's footsteps on turning collective intelligence into their advantage, like e-commerce websites, most noticeably Amazon.com, which delivers personalized content according to a user's habits of consumption. On the home page, users are presented with items that match their preferences, and they're given *recommendations* throughout the website. Using these tools as a marketing strategy, Amazon managed to beat the competition in both click-through and conversion rates [LSY03]. In fact, as of 2006, 35 percent of Amazon's sales were a result of recommendations [Mar06].

The success of *recommendation systems* drove the emergence, popularity and success of many other websites using them, such as *Last.fm*, *StumbleUpon* and *Netflix*.

Last.fm, a music service that learns what its users like and recommends more music, videos and concerts [Las]. The service became very popular among music-lovers and was bought by CBS in 2007, for $280 million [Tim07].

StumbleUpon, a service that recommends webpages based on the pages that a user previously liked or disliked [Stub]. Users became very addicted to the service, constantly using it to discover new websites, which led to its success and acquisition for $75 million in 2007, by eBay [eBa07].

Netflix, the world's largest movie rental service, in which 60% of the rentals come from a recommendation system [Netb]. The recommendations are so crucial to their core business that they offer a $1 million prize to anyone who manages to improve their system's accuracy by 10% [Neta].

It can be said that there's too much of *everything* on the web, and that that's the reason why search engines are so popular. However, search engines promote a trial and error behavior, with users spending their time checking every result until they're satisfied. Most people try only the first few results, and rarely adjust their queries. [Nie08].

Recommendation systems are much more personalized than search engines, and promote the discovery of new things—even without an explicit search [Por06]. And while they may be wrong at times, so can the search results. However, discovering interesting stuff is much more *fun* than adjusting keywords on search engines. This way, users can actually *enjoy* the searching and learning processes, which is a reason behind recommendation systems' success and popularity.

1.2 Project

This project focused on the design and development of *Content Blaster*—a recommendation system for anything that can be accessed via a URI, but not as a traditionally closed application or product. Instead, anyone is able to build applications on top of the system, taking advantage of the data that it provides and gathering more users in the process. Content Blaster is a *proof of concept*. It was created from scratch—a base system which can be further improved and expanded—and it's by no means a finished product.

This project was developed during eighteen weeks at the Fraunhofer headquarters in Portugal, at the AICOS research center. Fraunhofer is a German non-profit organization with more than 80 research centers and 50 institutes in Germany, which creates research-based products for the industry, the services sector and public administration [Fraa]. The Fraunhofer AICOS institute started in May 2008 as a joint project between Fraunhofer and the University of Porto, and develops market-oriented solutions based on research and development in the area of assistive information and communication technologies—creating solutions that lower the technical and financial barriers that usually hamper people from using them [Frab].

1.3 Motivation and Objectives

The growth in 3G access, smartphone and netbook sales is a clear indicator that an increasing number of people want to remain connected to the Internet constantly [NPD09, IT 09, USA09]. As mentioned before, with users being both consumers and producers, it's expectable that the web won't stop growing anytime soon—users need a way to manage the information overload on every Internet-capable device they have.

The aforementioned popularity of recommendation systems coupled with the ever-increasing number of mobile and desktop internet users creates a gap that this project attempts to fill, providing a way for users to have a recommendation system everywhere they go, in every device they have. The recommendation system shouldn't be a single application or product, but a platform capable of learning what its users' preferences are.

Content Blaster wants to be a personalized content provider for its users, where they take the role of passive spectators that just like or dislike what they're consuming, be it on

a smartphone, netbook, tablet PC, set-top box or a regular desktop or laptop computer.

These are the objectives of this project:

1. Research and document the state of the art on recommendation systems.
2. Design and implement a recommendation system that:
   - provides users with a way of voting how much they like a webpage;
   - recommends webpages to its users, using a state of the art algorithm;
   - is accessible over HTTP through a web-based application;
   - supports third-party applications;
   - is simple to implement and accurate within reasonable performance;
   - is easy to understand and simple to use by end-users.
3. Test the performance of the aforementioned system

1.4 Report Overview

The rest of this report is structured as follows.

Chapter 2: "State of the Art" starts by reviewing the various types of recommendation systems, comparing their strengths and weaknesses. It also provides an overview on two popular web-based recommendation systems.

Chapter 3: "Solution Specification" presents a brief and high-level overview on the proposed solution for the aforementioned objectives.

Chapter 4: "Technology" details the technologies used in Content Blaster and the reasons that led to their choice.

Chapter 5: "Design" gives the reader a detailed specification of the logical and physical architecture of the system. It also details some important design decisions.

Chapter 6: "Implementation" covers the most important implementation details of each of the components in Content Blaster.

Chapter 7: "Solution Evaluation" evaluates the performance of the recommendation algorithms used to generate recommendations in Content Blaster.

Chapter 8: "Conclusions and Future Work" reviews the project, drawing the necessary conclusions and pointing its benefits and limitations. It also provides some insights on which future developments have been considered.

Chapter 2

State of the Art

This section will provide an overview of recommender systems, detailing some types of systems and their variants, comparing their advantages and drawbacks. It will also describe how recommender systems are being used on extremely popular websites.

2.1 Recommender systems

Recommender or recommendation systems are present in many web-based applications of different nature. These systems are very complex, so it's fundamental to understand them properly. This section will provide an overview on:

- what a recommender system is and what *concepts* are inherent to them
- three main *types* of recommender systems, their advantages and disadvantages
- some recommendation *algorithms* that serve Content Blaster's purpose

2.1.1 Definition

A recommender system is one that uses the opinions of a large group of users to the purpose of helping them to identify content that may be of interest to them—a task that would be very difficult without the system, due to the overwhelming size of available content pool [HKTR04].

Although recommender systems can vary much from one another, there are a number of essential concepts that exist in all of them: *users*, *items*, *ratings*, *predictions* and *recommendations*. As these concepts represent the entities that play the main role in recommender systems, it's essential to understand them properly.

User. Someone using the system. Users must be registered so their data can be tracked.

Item. An individual unit collected by the system. Items can be songs or artists, books, webpages, people, or any commercial product—depending on the system's specific objectives and features. Users have some kind of interest in the content that motivates them to use the system—be it reading, listening, watching, playing, buying or selling.

Rating. The expression of an *user*'s preference on an *item*. The rating is the essential piece in recommender systems, and its usage fundamentally determines how the system will behave. Ratings may be explicit—when the user consciously gives feedback on the item—or implicit—if the rating is inferred from user behavior, such as purchase history on an e-commerce website like Amazon.com [BHK98]. If the rating is explicit, it can be presented to the user on a discrete numeric scale, e.g. from 1 (bad) to 5 (good), or in a qualitative one, such as *thumbs up* or *thumbs down*, like StumbleUpon [BHK98, Stub, Stuc].

Prediction. The predicted value of a *rating* for an *item* that an *user* hasn't rated yet. This value lies on the same scale of the *rating* [SKKR01].

Recommendation. A list of the top-N items that the system predicts a user will like the most. These are unrated items with higher predicted value for a specific user.

Recommender systems work by calculating *predictions* based on the actual ratings in the system. The *recommendations* for a user are the items with higher predicted value for that user [AT05].

2.1.2 Types

Even accounting for the specific characteristics of each recommender system, it's commonplace to categorize them according to one of the following groups [AT05]:

- Content-based systems
- Collaborative filtering systems
- Hybrid systems

Each one of these groups will be discussed in the next few sections.

2.2 Content-based systems

Content-based systems recommend *similar items* to those that the user liked in the past. Although this definition may seem to fit every type of these systems, that's untrue—this specific type of system uses the same techniques used in the fields of *information*

retrieval and *information filtering* [BS97, AT05]. That is to say that the system's recommendations are based on matching *textual content* about a set of items with the users' preferences [IdGL+08].

Such systems gather *content* about its items—a set of attributes, such as extracted keywords or descriptions characterizing the item [AT05, IdGL+08]. In order to generate recommendations, this content is matched with each *user's profile*—the preferences and interests of that user, based on the features of the items the user has rated highly in the past [BS97, AT05].

Because there are many variants of content-based systems and for the sake of comparison, it's important to define a *pure content-based recommender system* as

> "(...) one in which recommendations are made for a user based solely on a profile built up by analyzing the content of items which that user has rated in the past." [BS97]

Pure content-based recommender systems exhibit three main shortcomings: *shallow analysis*, *over-specialization* and *new users*.

Shallow analysis. Content-based systems are limited to the features that they are capable of extracting. This process works very well with textual data, but feature extraction from graphics, video or audio, while possible, is a very sophisticated problem [AT05, KT00]. Even with textual data, some features are very hard to extract. In web pages, factors such as user experience or aesthetics are completely ignored [BS97]. It may as well be impossible to differentiate between a well-written and a badly written newspaper article on the same topic [SM95].

Over-specialization. Because the system only recommends items according to the users' profile, the user is restricted to be recommended with items similar to those he already rated [BS97]. This may prevent *serendipitous discoveries*—the kind of discoveries that the user wouldn't make by himself [SM95]. It also may be problematic for the user, as he's subject to get multiple news articles describing the same event [AT05]. Recommender systems should promote diversity in its recommendations, not homogeneity—serendipity is an important factor, as it may give the user "*important intellectual leaps of understanding*" [IdGL+08].

New users. As the system creates the user's *profile* based on the items he rated highly, the user has to rate a considerable amount of items before he can be presented with good recommendations. Because of that, the initial recommendations for a new user may be unreliable [AT05].

2.3 Collaborative filtering systems

Collaborative systems or collaborative filtering systems don't take into account the content of the items in the system [GSK+99]. Instead of recommending items because they're similar, the system recommends items that *users with similar preferences* have liked [AT05]. The system starts by creating a set of *neighbors* for each user—those whose taste is similar to that of the user—and then recommends items that those *neighbors* have rated highly in the past [BS97, SKKR01]. Collaborative filtering makes possible to manage many types of content and to recommend items that are completely dissimilar to those the user has previously liked, effectively overcoming the *shallow analysis* and *over-specialization* problems of content-based systems [AT05, BS97].

Collaborative systems use very complex recommendation algorithms, and have many variants, some of which will be properly detailed in the next few sections. For now, and for the purpose of analysis and comparison with other recommender systems, it's important to define what a *pure collaborative filtering recommender system* is:

> "(...) A pure collaborative recommendation system is one which does no analysis of the items at all—in fact, all that is known about an item is a unique identifier. Recommendations for a user are made solely on the basis of similarities to other users." [BS97]

Although pure collaborative systems overcome the main problems of content-based systems, they have their own shortcomings: *scalability*, *sparsity* and *cold-start*.

Scalability. In collaborative filtering systems, comparing a large number of user profiles takes a considerable amount of time [LJB01]. Even when recommending items to a single a single user, a recommender system may exhibit problems if the number of items is very large [SKKR01]. It's possible to improve performance and scalability by reducing the data size—but at the cost of recommendations' quality [LSY03].

Sparsity. In a recommender system, there may be a very large number of both users and items. In this case, even with a large number of ratings, the user-item matrix may be very *sparse*—that is to say that there are very few ratings considering the possible number of ratings if every user rated every item [SKKR01, HCZ04]. Sparsity may cause very low similarity between two users, leading to a diminished number of recommendable items, which in turn renders collaborative filtering almost irrelevant [HCZ04, BS97]. The sparsity problem is the biggest shortcoming of collaborative filtering systems [HCZ04].

Cold-start. The cold-start situation is closely related with *sparsity*, but in the special case of a new user or item entering the recommender system [BH04]. In such case, the user or item doesn't have any known rating information. As collaborative systems

rely only on users' ratings, an item which hasn't yet been rated by a substantial number users won't be recommended frequently. The same is true for users, because the system must have a sufficient number ratings from that user to provide him with reliable recommendations [AT05]. This is the last problem is the same that content-based systems have with *new users*.

As mentioned, pure collaborative filtering have three key advantages over content-based systems [HKR02]:

1. They support items which content is difficult to analyze
2. The ability to recommend items based on their quality or taste
3. They can provide the user with serendipitous recommendations

Even with the aforementioned *scalability*, *sparsity* and *cold-start* problems, collaborative filtering became the most successful breed of recommender systems [SKKR01]. They may use a great variety of recommendation algorithms that have been traditionally classified as either *memory-based* or *model-based* [BHK98]. The rest of this section will discuss these algorithms. It will also cover a different class of algorithms, known as *item-based* [SKKR01].

2.3.1 Notation

This report will use a consistent notation to present the multiple metrics and algorithms used in recommender systems, as follows.

U is the set of all users

I is the set of all items

I_u is the set of items rated by user u

R is the set of all ratings

$r_{u,i}$ is the user u's rating on item i

$\bar{r}_u$ is the user u's average rating value

2.3.2 Memory-based

Memory-based algorithms operate over the *entire user database* to compute predictions *on the fly*. These algorithms create a set of *neighbors* for each user [BHK98]. Such *neighborhood-based* methods compute predictions based on the ratings of a user's closest neighbors. These algorithms can be decomposed in three steps [HKBR99, BKR07]:

1. Computing user similarity
2. Selecting neighbors
3. Generating predictions

Computing user similarity

The first step is to compute the similarity between users, so the *neighbors* can be selected. This step uses the ratings as the input values and creates a similarity value between pairs of users [BKR07]. The two most widely-used similarity measures are the *Pearson correlation coefficient* and the *cosine similarity*.

The *Pearson correlation coefficient* was introduced by [RIS+94] and computes the correlation between users u and v as defined in Equation 2.1.

$$sim_{u,v} = \frac{\sum_{i \in I_{uv}} (r_{u,i} - \bar{r}_u)(r_{v,i} - \bar{r}_v)}{\sqrt{\sum_{i \in I_{uv}} (r_{u,i} - \bar{r}_u)^2 \cdot \sum_{i \in I_{uv}} (r_{v,i} - \bar{r}_v)^2}} \tag{2.1}$$

Where I_{uv} denotes the set of items rated by both user u and v.

Using *cosine similarity*, also called *vector similarity* [HKBR99], users u and v are treated as m-dimensional vectors, where $m = |I_{uv}|$, and their similarity is measured by computing the cosine of the angle between them [BHK98, AT05], as defined in Equation 2.2.

$$sim_{u,v} = cos(\vec{u}, \vec{v}) = \frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\|_2 \times \|\vec{v}\|_2} = \sum_{i \in I_{uv}} \frac{r_{u,i} \cdot r_{v,i}}{\sqrt{\sum_{j \in I_{uv}} r_{u,j}^2} \cdot \sqrt{\sum_{j \in I_{uv}} r_{v,j}^2}} \tag{2.2}$$

Of these two metrics, *Pearson correlation coefficient* [HKBR99] is the most used in collaborative filtering recommender systems, and was proved by [BHK98] to outperform *cosine similarity*.

Selecting neighbors

Instead of using the entire neighborhood of a user to serve as predictors during the next step, collaborative filtering systems usually select just a subset of these neighbors, both for *performance* and *accuracy* reasons. As the number of users in these systems grow, *performance* becomes an issue, and very similar neighbors are much more valuable than distant ones, providing more *accurate* predictions [HKBR99].

There are two techniques to filter neighbors: *similarity threshold* and *nearest-K neighbors* [ZP07].

Correlation thresholding selects neighbors with a similarity value greater than a given threshold value. However, choosing this threshold may be a problem, as an high

threshold may select a very small neighborhood and a low threshold may result in a very large one, rendering almost useless the purpose of selecting a subset of the neighborhood [HKBR99]

Nearest-K neighbors is a simple technique that selects the top K neighbors in terms of similarity value [ZP07].

Between these two, *nearest-K neighbors* was found by [HKBR99] to be the best option, providing good results without major performance problems.

Generating predictions

The final step to provide recommendations is generating predictions from the *selected neighborhood*. The simplest way to generate is through a simple *weighted sum* over the neighbors' ratings, using the similarity value as the weight—so the most similar neighbors become predominant [AT05].

In order to generate the prediction $p_{u,i}$ for a user on a particular item, *weighted sum* uses the set of user u's neighbors who rated that item, denoted $\hat{U}_u$.

$$p_{u,i} = \frac{\sum_{v \in \hat{U}_u} sim_{u,v} \cdot r_{v,i}}{\sum_{v \in \hat{U}_u} sim_{u,v}} \tag{2.3}$$

One major drawback of *weighted sum* is that it doesn't take into account how users rate their times—each one may have his own way of using the rating scale. So, to circumvent this shortcoming, instead of using the ratings of a *neighbor*, the *deviation from mean* or *adjusted weighted sum* uses their deviation from the *neighbor's* average rating value to compute the prediction [AT05, HKBR99].

$$p_{u,i} = \bar{r}_u + \frac{\sum_{v \in \hat{U}_u} sim_{u,v} \cdot (r_{v,i} - \bar{r}_v)}{\sum_{v \in \hat{U}_u} sim_{u,v}} \tag{2.4}$$

Deviation from mean is clearly a more complex algorithm than *weighted sum*, but [HKBR99] found it to be more accurate.

Summary

Memory-based collaborative filtering is fairly simple, and has a significant number of options in terms of which algorithms to pick for a recommender system. There are some improvements on these systems that weren't presented, such as *default voting*, *inverse user frequency* and *case amplification* [BHK98]. For instance, *case amplification* operates on the similarity value between two users, setting a new $sim'_{u,v}$ to be used when calculating

predictions.

$$sim'_{u,v} = \begin{cases} sim^{\rho}_{u,v} & \text{if } sim_{u,v} \geq 0 \\ -(-sim^{\rho}_{u,v}) & \text{if } sim_{u,v} < 0w \end{cases}$$

However, as techniques such as these are just small *improvements* on established algorithms, further research on these topics is out of the scope of this project and, therefore, this report.

2.3.3 Model-based

In contrast to *memory-based* algorithms, *model-based* algorithms use the ratings set to build a *model* of user preferences using statistical and machine learning techniques, and then infer the predictions from this model [AT05]. This model can be built using many different techniques, but this section will focus only on the model proposed by [BHK98], where the prediction is viewed from a probabilistic perspective.

$$p_{u,i} = E(r_{u,i}) = \sum_{v=0}^{n} v \times \Pr(r_{u,i} = v | r_{u,j}, j \in I_u) \tag{2.5}$$

It's assumed that the ratings values are integers between 0 and n. The probability expression is the probability that a user u will give a particular rating v to item i, based on his previously rated items I_u. In his research, [BHK98] proposed two models for the probabilistic model: *cluster models* and *Bayesian networks*.

Cluster models' idea is that there are certain types of users with similar preferences, and it works by clustering the users into classes. Given a particular class, the ratings are independent. Parameters such as probability of class membership and the conditional probability of ratings given a class are learned from the available data.

The other model is a *Bayesian network* where *items* are the *nodes* and its *states* are the possible *rating* values, including *no rating* in a domain without a default rating [AT05]. The algorithm created by [BHK98] searches through multiple model structures for the dependencies of each item, resulting in a network where items' parents serve as *predictors*. This technique computes predictions based on a decision tree for how a user will rate a specific item.

These probabilistic models are an example of a wide array of choices for model-based algorithms, such as latent semantic analysis, statistical models, decision trees, aspect models, Bayesian models, linear regression, maximum entropy models, neural networks or singular value decomposition [LM05, BH04, AT05].

2.3.4 Comparison of memory and model-based algorithms

Memory-based collaborative filtering algorithms are clearly much simpler than its model-based counterpart, and produce reasonable recommendations as well [PHLG00]. They are easier to use as they have less parameters that need tuning and new data can be easily and incrementally added—reasons that made them more prevalent among recommender systems [BH04].

However, model-based systems have better accuracy, performance and efficiency when generating recommendations, and they handle sparsity better than memory-based systems [PHLG00, BH04, AT05].

Memory-based systems use a very large database to generate their recommendations, while the model produced by the Bayesian network is much smaller—but building this model is computationally expensive and may need to be performed when new data is added to the system [BHK98, PHLG00]. Despite being very fast and small, the model is inappropriate for systems in which user preferences changes rapidly [SKKR01].

When comparing memory and model-based algorithms, [BHK98] found that *Bayesian networks* with decision trees at each node and *deviation from mean* using the *Pearson correlation coefficient* outperformed other algorithms in most of their tests. They didn't prefer one over the other, and explained that such choice must take other factors into account, such as the nature of the database and the application.

2.3.5 Item-based

As mentioned before, collaborative filtering systems traditionally explore the *similarity between users* to generate predictions and have two major drawbacks: *scalability* and *sparsity*. From the many techniques focused on overcoming these problems, *item-based collaborative filtering* [SKKR01] is particularly interesting and is being used successfully on Amazon.com [LJB01, LSY03].

The idea behind this technique is that users will like *items which are similar* to those they have liked in the past, and will avoid those which are similar to the ones they have disliked. This train of thought doesn't require the analysis of a user's neighborhood when he requests a recommendation to the system. Instead, item-based collaborative filtering explores *similarity between items*, recommending similar items to those the user has liked before, instead of items that *similar users* have liked. Because the recommendation system doesn't analyze the user's neighborhood, it tends to generate recommendations much faster [SKKR01].

The rest of this section will present and compare two different algorithms: *traditional item-based collaborative filtering*, as proposed by [SKKR01] and the *Slope One* algorithm created by [LM05].

Notation

Some additional notation is required for *item-based* algorithms, as follows:

U_i is the set of users who rated the item i

U_{ij} is the set of users who rated both item i and j

$\bar{r}_i$ is the item i's average rating value

Item-based collaborative filtering

This technique was proposed by [SKKR01] and behaves much like memory-based algorithms, calculating item similarity values with either correlation or cosine similarity, but on the item space. The algorithm uses only a subset of each item's most similar items to generate the prediction.

Just as traditional memory-based algorithms, similarity between items can be calculated with *Pearson correlation coefficient* or *cosine similarity*, as defined by Equations 2.6 and 2.7, respectively.

$$sim_{i,j} = \frac{\sum_{u \in U_{ij}} (r_{u,i} - \bar{r}_i)(r_{u,j} - \bar{r}_j)}{\sqrt{\sum_{u \in U_{ij}} (r_{u,i} - \bar{r}_i)^2 \cdot \sum_{u \in U_{ij}} (r_{u,j} - \bar{r}_j)^2}} \tag{2.6}$$

$$sim_{i,j} = cos(\vec{i}, \vec{j}) = \frac{\vec{i} \cdot \vec{j}}{\|\vec{i}\|_2 \times \|\vec{j}\|_2} = \sum_{u \in U_{ij}} \frac{r_{u,i} \cdot r_{u,j}}{\sqrt{\sum_{v \in U_{ij}} r_{v,i}^2} \cdot \sqrt{\sum_{v \in U_{ij}} r_{v,j}^2}} \tag{2.7}$$

However, there's one fundamental difference between *user-based* and *item-based* algorithms. In the former, similarities are calculated *along* a single user, while the latter calculates similarity *across* users, who may use the rating scale in their own way. In order to circumvent this problem, [SKKR01] proposes the *adjusted cosine similarity* measure, which subtracts the user average rating from each one of his ratings.

$$sim_{i,j} = \sum_{u \in U_{ij}} \frac{(r_{u,i} - \bar{r}_u)(r_{u,j} - \bar{r}_u)}{\sqrt{\sum_{v \in U_{ij}} (r_{v,i} - \bar{r}_v)^2} \cdot \sqrt{\sum_{v \in U_{ij}} (r_{v,j} - \bar{r}_v)^2}} \tag{2.8}$$

This new measure alone results in much better accuracy compared to pure *cosine* or *correlation-based* similarity [SKKR01].

Just like traditional memory-based algorithms—where most *user pairs* don't have a common item—in item-based collaborative filtering most *item pairs* have no common user, so iterating through all item pairs is inefficient. In order to circumvent this problem, [LSY03] proposed a way to find which pairs of items were rated by a same user. Instead of iterating through all possible item-item pairs, which would take n^2 iterations for n

items, he suggests another technique. For each item, it checks users who have rated it and find out other items they have rated. These items share a user with the current item, thus constituting valid item pairs who share at least one user. These pairs are recorded for later use. After using this technique, the algorithm can now calculate the similarity for the recorded pairs only. This optimization wasn't used by [SKKR01] but it's an improvement to be considered.

When computing the predictions, [SKKR01] considers two techniques: *weighted sum* and *regression*. The *weighted sum* technique is identical to the one used in memory-based algorithms, but on the item space. The *regression* technique uses a *linear regression model*. However, as regression is overall less accurate than *weighted sum* and always less accurate than *user-based* algorithms [SKKR01], and for the sake of comparison with *Slope One*, it won't be part of this discussion and comparison with the Slope One algorithm. The *weighted sum* calculates the prediction $p_{u,i}$ based on item i's similar items that the user u has rated, denoted as $\hat{I}_i$.

$$p_{u,i} = \frac{\sum_{j \in \hat{I}_i} sim_{i,j} \cdot r_{u,j}}{\sum_{j \in \hat{I}_i} |sim_{i,j}|} \tag{2.9}$$

Just like the memory-based approach only needs a subset of the user's neighbors, this algorithm only needs a fraction of the item similarities, so [SKKR01] explored a different approach to his algorithm. Instead of storing all item-to-item similarity values, the algorithm creates a model with the top-l most similar items of each item, thus naming l the *model size*. Using a *model size* smaller than the maximum possible number produces less accurate recommendations, but delivers a much improved performance [SKKR01]. When computing the prediction for a user on a specific item, the algorithm only uses the top-k *neighbor items* rated by that user, where $k < l$. The model and neighborhood sizes aren't fixed, and may be tuned for optimized performance and accuracy.

Although item-based algorithms are a complete new approach to recommender systems, they're most commonly classified as model-based systems [SKKR01, PPM+06, DK04], because of their *model learning step*, the item-to-item similarity computation, which is done offline. However, as the algorithm proposed by [SKKR01] is so similar to traditional memory-based algorithms, there are a few researchers who classify item-based collaborative filtering as a memory-based technique [AT05, BH04].

Item-based collaborative filtering is a novel type of collaborative filtering, and just like there are many user-based algorithms, item-based collaborative filtering can take many forms—the aforementioned algorithm is just one of those. The next section will discuss another one, the *Slope One* algorithm, and compare it with both *item-based* and *user-based* algorithms.

Slope One

The Slope One algorithm was created by [LM05] and constitutes another approach at item-based collaborative filtering. This technique is easier to implement, provides accurate results and is divided in two parts—*computing deviations* and *generating predictions*.

Much like the previously mentioned algorithms, it computes a numeric value corresponding to a relation between users or items. While in most cases this value denotes *similarity*, *Slope One* computes the average *deviation* of an item i with respect to item j.

$$dev_{j,i} = \sum_{u \in U_{i,j}} \frac{r_{u,j} - r_{u,i}}{|U_{i,j}|} \tag{2.10}$$

It can be derived from Equation 2.10 that $dev_{x,y} = -dev_{y,x}$. As such, the item deviation matrix, which contains all the item-to-item deviations, is *skew-symmetric*: $M^T = -M$.

Using the deviation value, a way to predict the rating of a user on a specific item given any of his ratings could be a simple $p_{u,i} = r_{u,j} + dev_{i,j}$ formula. To compute predictions for a user, the *Slope One* algorithm extends this formula to all of his rated items which have at least one deviant item.

$$p_{u,i} = \sum_{j \in D_{u,i}} \frac{dev_{i,j} + r_{u,j}}{|D_{u,i}|} \tag{2.11}$$

Where $D_{u,i}$ is the set of all items rated by user u that are *deviant* in relation to i, that is $D_{u,i} = \{j | j \in I_u \wedge i \neq j \wedge |U_{i,j}| > 0\}$. Since sometimes every item rated by a user has at least one deviant item, this approximation can be considered:

$$\sum_{i \in I_u} \frac{r_{u,i}}{|I_u|} \simeq \sum_{j \in D_{u,i}} \frac{r_{u,j}}{|D_{u,j}|}$$

Applying this approximation in Equation 2.11 makes the formula simpler, giving the basic *Slope One* formula, a simple *average of deviations*.

$$p_{u,i} = \bar{r}_u + \sum_{j \in D_{u,i}} \frac{dev_{i,j}}{|D_{u,i}|} \tag{2.12}$$

However, this formula doesn't take into account how many users have rated both i and j. This motivated [LM05] to create a variant, the *Weighted Slope One*. This variant is a *weighed average of deviations*, in which the uses the number of users that rated a common pair of items as the *weight*.

$$p_{u,i} = \frac{\sum_{j \in D_{u,i}} (dev_{i,j} + r_{u,j}) \cdot |U_{i,j}|}{\sum_{j \in D_{u,i}} |U_{i,j}|} \tag{2.13}$$

Summary

The item-based collaborative filtering developed by [SKKR01], using *weighted sum* with *adjusted cosine similarity*, produces better recommendations than the best user-based algorithm in terms of accuracy. It's capable of scaling as well, as as it pre-computes a model which size may be tuned to improve performance. Effectively, [SKKR01] created a novel, better approach to collaborative filtering, used successfully at very large recommender systems such as Amazon.com [LJB01].

In the case of *Slope One*, [LM05] compared both variants against the aforementioned *item-based* algorithm and an *user-based* algorithm using *weighted sum* of *Pearson correlation coefficient* similarities. All three variants of *Slope One* shown better recommendation accuracy than the other item-based algorithm created by [SKKR01]. Interesting enough was the fact that every *Slope One* algorithm proved worse than the *user-based* algorithm in terms of accuracy. This contradicts the results of [SKKR01], where his *item-based* algorithm proved to have better accuracy than the best *user-based* algorithm, which also used *Pearson correlation coefficient*. Although the trustworthiness of *Slope One*'s results may be up for discussion, the difference in terms of accuracy is very small, and *Slope One* shows an unprecedented simplicity.

Comparing both variants of the *Slope One* algorithm, *Weighted Slope One* shows a mere improvement of 1% in accuracy over the basic algorithm, but at the cost of a more complex formula.

2.4 Hybrid systems

Hybrid systems combine two or more types of systems, seeking to overcome one's drawbacks with the other's strengths, resulting in an unique system with less drawbacks [Bur02]. Hybrid systems can be made up either by combining separate systems, adding characteristics of one to the other, or creating a new unified system [AT05].

2.4.1 Combining separate systems

This type of *hybrid systems* implements separate recommendation techniques with the results of each one being combined by *weighting*, *switching* or *mixing*.

Weighting. A weighted hybrid recommender is one that uses multiple recommendation methods, and computes their outputs into a single one. The simplest way to do this is by a linear combination of the recommendation values [AT05], but there are some other variants of *weighted systems*. In past work, [CGM+99] used an hybrid of content-based and collaborative filtering, but gradually adjusted the weight factors on each system as their predictions are confirmed or disconfirmed by the user.

Although *weighted hybridization* brings together each technique's advantages, common weaknesses among techniques aren't avoided—using a content-based method and a collaborative filtering method wouldn't solve their common *cold-start user* problem [Bur02].

Switching. This type of system *switches* between recommendation methods, using some quality metric to make the choice of which to use among the available ones [AT05]. This technique can be sensitive to each technique's advantages and drawbacks, with the added cost of using what may be a complex and parameterized criterion and doesn't overcome their common drawbacks [Bur02].

Mixing. Mixed systems employ all the available techniques to produce recommendations and present them at the same time. This effectively promotes each technique's major advantages, as both recommendations are shown. The major problem with this approach is that because recommender systems usually require the ranking of recommendations or choosing a single best recommended item. In either case, mixed systems fall short and some other technique must be used.

2.4.2 Adding characteristics

Most commonly, this type of systems is based in collaborative filtering while also maintaining a content-based profile for each user. Collaborative filtering operates over the content-based profiles instead of commonly rated items [AT05]. Fab [BS97] and *filterbots* [SKB+98] are two example of this type *hybrid systems*.

Fab uses a content-based system to build a model of user profiles, and uses this *entire model* as the input for a collaborative filtering system, and user get recommendations for items that match a similar user profile or their own profile. Fab avoids the *sparsity* problem, as well as the *new item* problem. So-called "grey sheep" users, who have very specific preferences and don't fall into any group of users can get accurate recommendations as well [Bur02].

In past work, [SKB+98] developed a collaborative filtering system in which some users were *filterbots*—content-based filtering agents that rate items based on their content [GSK+99]. These agents rate items based on some criterion—the length of a document, the number of external references to it, or the spelling correctness—and act as pseudo-users in the system, helping users with better recommendations [SKB+98]. The *filterbots* technique is very simple, as collaborative filtering systems don't have to treat them differently—they are virtually identical to any other user. Later work by [GSK+99] proved that a system with multiple users mixed with multiple agents is the best overall scheme for a *hybrid system* using the *filterbots* technique.

2.4.3 Unified system

This type includes every other system that uses both systems in such a way that doesn't fit the previous types of *hybridization*. For example, the work of [BHC98] demonstrated better recommendations using both content and collaborative features in a single rule-learning system. Hybrid unifying systems are very complex and require deep research, so further discussion on these topics is out of this project's scope.

2.4.4 Summary

Hybrid systems appear in many forms, motivated by the weaknesses of both collaborative filtering and content-based systems. Hybrid systems have proved to provide more accurate recommendations than pure systems [AT05], but as many of them employ two fully operational techniques, they do require a considerable implementation effort. The only exception may be the *filterbots* technique, as adding these agents to an already existent collaborative filtering system is fairly easy and the agents may be adapted to match the system's characteristics [GSK+99].

2.5 Comparison of recommender systems

The recommender systems discussed in the previous sections exhibit advantages and shortcomings, with hybrid systems trying to bring each one's advantages together. Table 2.1 lists the most important advantages and drawbacks of content-based and collaborative filtering systems.

Table 2.1: Comparison of recommender systems

METHOD	ADVANTAGES	DRAWBACKS
Collaborative filtering	Independent of medium Serendipity	New users New items Sparsity
Content-based	Recommends recent items Doesn't need many users	Can't analyze some media New users Over-specialization

Both content-based and collaborative filtering exhibit problems when providing recommendations to *new users*, but content-based systems are immune to the *cold-start items* issue, because new items don't need many ratings to be recommended to someone. However, because content-based systems analyze content only, they're affected by *over-specialization*, in contrast with the *serendipity* of collaborative filtering that may actually surprise and engage its users. The characteristics of content-based systems also makes them dependent of their ability to extract features from the content, which sometimes may be impossible,

while collaborative filtering is independent to the medium of the item, while also being able to recommend and item based on quality or aesthetics. But their major shortcoming may be operating in *sparse environments*, where their accuracy is deeply affected, but this problem is less pronounced in *model-based* or *item-based* systems. Hybrid systems try to get the best of both approaches. They usually provide more accurate recommendations, but they're also more complex, in some cases implementing two fully operational recommendation algorithms.

2.6 Real-world recommender systems

There are a few web-based recommender systems that became very popular in recent years. This section presents two recommender systems for webpages: *StumbleUpon* [Stua] and *Digg* [Diga].

2.6.1 StumbleUpon

StumbleUpon is a free web-based service recommendation engine that helps people discover new websites. It's used through a toolbar installed on users' internet browsers, and has two fundamental concepts: *Like/Dislike* and *Stumbling*.

Like/Dislike. Users may mark any webpage as liked or disliked by themselves by pressing either the "thumbs up" or "thumbs down" button on StumbleUpon's toolbar.

Stumbling. The name given to the act of getting consecutive recommendations. This is derived of the "Stumble!" button that takes users to a recommended webpage when clicked.

After registering on StumbleUpon, a user must select any number of topics of his preference. These topics are grouped under categories, which can be found on Appendix B. While the core of the user experience is based on liking/disliking pages and *stumbling*, users may also:

- Comment and tag pages
- Add other users as friends and share pages with them
- Browse pages by topic, tag or media type (some particular topics represent media types, such as Photography, News or Videos)
- Stumble by topic
- Search for pages
- Stumble based on a search query

Although the StumbleUpon team doesn't provide detailed explanation on the system's underpinnings, they have published a brief overview on their recommendation engine [Stuc]. StumbleUpon recommends pages using three techniques, based on the ratings of the user or their friends: *topics*, *socially endorsed pages* and *peer endorsed pages*.

Topics are provided via a classification engine that operates on users' ratings on webpages.

Socially Endorsed Pages are pages recommended by friends. This may refer to pages *shared* by friends.

Peer Endorsed Pages are endorsed by similar users. This is the part where StumbleUpon provides more detail, saying that "*rating websites updates a personal profile (weblog) and generates peer networks of websurfers linked by common interest*"

StumbleUpon uses all three techniques when recommending webpages, being a complex *hybrid system*, but they provide no detail on how much each technique weights on the final recommendations. Although StumbleUpon is an *hybrid system*, every information available makes it feasible to conclude that the system doesn't use any content-based technique.

It's important to notice that StumbleUpon doesn't use a recommender engine as part of a feature—the only purpose of the system is to discover new webpages. StumbleUpon recently hit an impressive number of seven million users [Kir09], is ranked 425th on the Alexa Traffic Rank [Alec], and has been growing in popularity, reaching between 4 and 6 million unique visitors per month in the January–May period of 2009 [Comb].

StumbleUpon is an example of the popularity of recommender systems and a proof that recommender systems are capable of scaling, given proper optmization.

2.6.2 Digg

Digg is a *social news* website, where content is submitted and promoted by users [Digb]. It works in a very simple way:

1. **A user submits a webpage to Digg**. Inside Digg, pages are known as *stories* and must belong to a single topic.

2. **Digg users vote on stories** based on some criterion—most commonly, if they like them. An user can only vote on a page once, and a vote is known as a *digg* inside the Digg community. Many website owners add Digg *badges* to their pages. These badges show the number of *diggs* of that page and have a button to *digg it* that particlar page, so it's easier for vistors to *digg* without ever going to Digg.com itself.

3. **The stories with the most diggs are promoted to Digg's frontpage**. The number of *diggs* of a story is a measure of how popular that it is, and being on the Digg frontpage is a huge source of new visitors for any website.

As the popularity of Digg increased, the number of new stories being submitted each day started to grow up to a enormous amount, which motivated Digg to create a recommendation engine so users could filter new stories without being overwhelmed [Ros09]. This recommendation was further detailed in [Kas]. Digg's recommender is a pure user-based collaborative filtering system that computes a correlation coefficient between users and selects users' neighborhoods with correlation thresholding. The correlation is only based on the last thirty days of activity and is computed for user u as

$$corr_{u,v} = \frac{Diggs_{u,v}}{Diggs_u}$$

where $Diggs_u$ is the number of *diggs* of user u and $Diggs_{u,v}$ is the common number of *diggs* among users u and v. These correlations are computed both at a global and a topic level (Digg's topics are listed in Appendix C). This system is very exposed to users, as they known why a particular *story* was recommended and can view their neighbors and their *dugg stories* [Ros09].

When developing the system, scaling was constantly considered, as Digg has dozens of thousands of new *stories* on a daily basis. Everytime a user *diggs* a story, the correlation coefficients between him and every other user are recomputed, which was only possible with the creation of a custom database management system, tailored to the specific needs of Digg [Ros09].

Digg is even more popular than StumbleUpon, with more than 30 million unique visitors per month in the January–June period of 2009 [Coma], and it's ranked 165th on the Alexa Traffic Rank [Aleb].

2.6.3 Summary

Both Digg and StumbleUpon proved that recommender systems with simple concepts can be successful and popular, helping users to filter through the ever-increasing amount of content available on the web. Although scaling is a very serious problem in large systems, they have both proved capable of handling it properly.

2.7 Chapter summary

This chapter reviewed the work on the field of recommender systems, fundamental to the planning, design and development of Content Blaster. It started by presenting both *content-based* and *collaborative filtering* systems.

Content-based techniques have proved inappropriate to use when content analysis is difficult or impossible. On the other hand, collaborative filtering strives in the same conditions but has its own drawbacks: *sparisity* and *scalability*. Model-based techniques try to address these shortcomings. They provide better recommendations than memory-based algorithms, even in sparse environments, but they're very complex when compared to memory-based systems' simple and straightforward design. All of these techniques are user-based, but an item-based algorithm proved better than the best user-based technique without being overwhelmingly complex. This algorithm was compared to another item-based technique, Slope One. They both provide reasonably good recommendations, but the latter has a significantly simpler implementation.

The section also presented *hybrid systems*, which try to bring together the best of the previous ones. They usually succeed but often require the implementation of two recommender systems, which means an increased development effort. The only exception are *filterbots*, agents that act as pseudo-users, rating items based on their content.

Finally, two popular real-world web-based systems were presented, Digg and StumbleUpon. Both systems proved capable of handling the scalability problem, and their popularity and success constitute an extra motivational factor for this project.

Chapter 3

Solution Specification

This chapter presents the solution defined to address the objectives enumerated in Section 1.3, based on the research documented on Chapter 2.

3.1 Overview

Content Blaster is a web-based recommender system for webpages, accessible via a web browser. Each page may have any number of *tags* that are automatically fetched from other websites, but Content Blaster's users may add their own.w

Content Blaster is an hybrid recommender system, using the *Slope One* collaborative filtering algorithm combined with *filterbots*. Each *filterbot* corresponds to a tag in the system, and automatically likes pages that were tagged with the tag it represents.

Using just their browser, users may like, dislike and *tag* any webpage, and get recommendations from Content Blaster. The system also provides third-party access via a simple API, so it can be extended to other applications and devices.

The next sections will present the system with more detail, explaining the reason behind the choices that were made when designing Content Blaster.

3.2 Recommender System

As mentioned before, Content Blaster is an hybrid recommender system, using the *Slope One* collaborative filtering algorithm coupled with simple *tag*-based *filterbots*. This decision is the result of the research on recommender systems, considering Content Blaster's objectives and constraints.

As mentioned in Section 1.3, Content Blaster should "provide users with a way of voting how do they like a webpage". Nowadays, users not only spend their time reading

documents, but they're constantly watching videos, browsing photos, playing games and listening to music on websites. The web is a platform that supports multimedia content, and not every webpage is mostly composed of text. With that in mind, *collaborative filtering systems* were easily preferred over *content-based* ones, because of their ability to operate regardless of the item's content. Both Digg and StumbleUpon support any type of media, and as far as it's known, they don't analyze their webpages' content either.

The three types of collaborative filtering techniques were considered for Content Blaster. Based on the research presented in Chapter 2, they were compared in terms of *dealing with sparsity*, *recommendation quality*, *difficulty of implementation* and *scalability*. Some of these metrics are fundamental to any recommender systems, whether others have particular interest in collaborative filtering systems, as it's the case of Content Blaster.

Dealing with sparsity. Sparsity is the major drawback of collaborative filtering techniques, and the *memory-based* methods are the most affected. Both *model-based* and *item-based* systems show better results than *memory-based* ones under a sparse environment.

Recommendation quality. Model-based schemes have proved more reliable than memory-based schemes, but they're much more complex. The item-based algorithm of [SKKR01] also proved to outperform the best user-based algorithm in both performance and accuracy.

Difficulty of implementation. Despite all their weaknesses, memory-based methods are very easy to implement, specially when compared to their model-based counterparts. As mentioned before, *item-based* and *memory-based* methods are very similar in terms of implementation.

Scalability. Scalability is a serious problem *in every* recommender system. Model-based and item-based techniques provide fast recommendations, but at the cost of an expensive *learning phase*. In item-based systems, tuning the model size may help to optimize this learning phase. Memory-based schemes are more affected by scalability problems, but it's important to remember that both Digg and StumbleUpon have proved capable of overcoming this problem, handling millions over users everyday.

The technique chosen for Content Blaster was item-based collaborative filtering, as it is easy to implement while also providing good results even in sparse environments. It's important to remember that one of the objectives for Content Blaster was to be "accurate within reasonable performance"—a minor gain in accuracy may not be worth a major sacrifice in simplicity or scalability.

The choice of which item-based algorithm to use was the most difficult to make. The results of [LM05] contradicted with [SKKR01] and it's debatable if a relatively new algorithm as Slope One is effectively better than traditional item-based algorithms. However,

both of them are on pair in terms of accuracy, as the reported improvement of Slope One was minor. Taking this very little difference in accuracy into consideration, Slope One's simplicity in terms of design and implementation is remarkable, being the factor that favored it over traditional item-based collaborative filtering.

On Content Blaster, the algorithm runs periodically to provide and store item-to-item deviation values. After this step, it also computes and stores predictions for every user—the system doesn't compute recommendations in real-time. This was done for performance reasons, because a shared computer was used for deployment. The recommendation given to a user is one of the top ten predicted items, chosen randomly, so the user won't always get the same recommended item if he doesn't rate it.

3.3 Tagging

In Content Blaster, users may add *tags* to a page when they like or dislike it. This is done to alleviate the *cold-start* problem with the help of *filterbots*, which represent pseudo users in the system. Instead of analyzing the page's content—which is a difficult task considering the multimedia support in the web today—the system uses *tags* added by its users, that help to categorize and describe that page's content. As tags are optional in order to make the rating process easier and faster for users, Content Blaster also fetches tags from other websites that aggregate and organize a huge number of webpages: *Digg*, *StumbleUpon* and *Delicious*.

Digg. As mentioned before, Digg groups their stories into topics. Content Blaster users the name of the topic of a page as the tag

StumbleUpon. StumbleUpon works like Digg, but users may also add tags to their pages. Both topics and tags are extracted from StumbleUpon.

Delicious. Delicious is a *social bookmarking* website, where users can "save, manage and share web pages from a centralized source" [Del, Yah]. Delicious' users aren't forced to choose one of the predefined topics to categorize their page—they may add their own tags. Content Blaster uses those tags.

When a page is added to Content Blaster, the system searches these three other websites for the newly added page. If it's already there, it fetches the topics and tags used on those systems to categorize the page, and uses that information to tag the page on Content Blaster. When any number of tags is added to a page in Content Blaster, the bots which represent those tags automatically like that same page. If a new tag enters the system, the corresponding bot is automatically created.

Using filterbots helps to alleviate the *cold-start* problem, because a new item will automatically have a number of *filterbots* that like it. This means that new users, which

only like one or two items, will get recommendations from the bots which also liked that item, and new items will get recommended faster [PPM+06]. It will also helps to deal with *sparsity*, as two items which have a common tag will have at least one common user [SKB+98].

This technique is relatively simple to use, but as a result, the number of bots can be very large. Considering the popularity and size of Digg, StumbleUpon and Delicious, together with user-added *tags*, it's expected that most items will have *tags*, effectively helping to address some of the traditional collaborative filtering techniques' drawbacks.

3.4 Voting scheme

The voting scheme wasn't subject to much discussion. A *like/dislike* scheme was easily preferred because it's much simpler for users than a discrete numeric scale, as they only have to ponder between two options. This voting schema may be used in Slope One, as it was confirmed by its inventor to be scale invariant (see Appendix D). Non-numeric voting schemes are used with success in both Digg and StumbleUpon.

3.5 Usage

Content Blaster is used through two buttons added to a regular browser: *Like/Dislike* and *Blast me!*.

Like/Dislike Opens a small window in which users can add tags to the page and mark it as liked or disliked.

Blast me! Redirects the user to a recommended page.

These buttons may be displayed using one of three techniques: *installed toolbars*, *frame toolbars* and *bookmarklet buttons*.

Installed toolbars. The website requires users to install a software *add-on* that adds functionality to their browsers. This method doesn't add content to the page—the toolbar becomes part of the browser's interface. This is the preferred method of StumbleUpon.

Frame toolbars. The toolbar is displayed on the website, with the actual page being seamlessly inserted on a frame below the toolbar. This method tries to overcome the need of users having to install software. However, as frames replacee the actual page's URL with another one, they break the functionality of the back button and may pose a problemf for search engines [Nie99]. This is the preferred method of Digg.

Bookmarklet buttons. Bookmarklets are just like regular bookmarks, but they execute some other function than opening a page when clicked. They can be added to an existing browser toolbar by dragging them from any page to the toolbar, where they appear as buttons. This doesn't require installing software, overcomes the problems of using frames and may also be used in some mobile browsers. This is the preferred method of Content Blaster.

3.6 Third-party support

Content Blaster was designed to support third-party applications via a simple *application programming interface*. Any developer may incorporate Content Blaster's features into any application, using the user's *security token* for authentication. Content Blaster supports the following operations:

- Liking, disliking and tagging webpages
- Getting a list of recommendations

3.7 Chapter summary

This chapter started by presenting a brief overview of Content Blaster. The Slope One algorithm was chosen taking into consideration factors such as *dealing with sparsity*, *recommendation quality*, *difficulty of implementation* and *scalability*, as it provides accurate recommendations with an extremely simple design and good performance.

The following section presented the motivation and reasons behind the choice of using *filterbots*, agents that improve recommendation quality by acting as users which rate pages based on their tags (added by both users and third-party services).

The rest of the chapter explained the solution used for the voting scheme used in the system, the reasonably simple and small feature set available to both users and third-party developers, and how they can use those features.

Chapter 4

Technology

Content Blaster is a web-based application developed in Ruby on Rails, a Ruby framework. It uses a MySQL-powered database, and runs on a Apache server with the Passenger module. This chapter will explain these technologies and why they were chosen.

4.1 Ruby

Ruby is a dynamic and object-oriented language used for multiple purposes, created by Yukihiro Matsumoto, also known as *Matz* [FM08]. Matz wanted to create a "*language that was more powerful than Perl, and more object-oriented than Python*", and his philosophy for Ruby is better explained with a quote of his own:

> "Ruby is designed to make programmers happy"

Ruby was inspired by languages such as Lisp, Smalltalk and Perl, and its core characteristics and features are [Rub, FM08]:

Open source. Ruby's license allows anyone to use, copy, modify or distribute it.

Pure object-oriented. In Ruby, *everything* is an object, including classes, modules and data types—even numbers, booleans and null values which in other object-oriented languages are known as *primitives*. An object's properties are known as *instance variables* and actions are *methods*.

Flexible. Not only it features dynamic typing, but also very powerful reflective and metaprogramming capabilities. Ruby doesn't restrict what a programmer can do. Any part of Ruby code can be removed, redefined or extended, even at *runtime*. This isn't only true for a programmer's own code, but also for core Ruby classes such as Object and String.

Multi-threading with *green threads*. Ruby can run multiple threads, but without relying on the operating system capabilites. This means that Ruby threads can only use a single processor.

Automatic memory management. Like other languages such as Java and contrary to C, developers don't manage the program's memory usage.

Portable. Ruby can run on most operating systems and platforms, such as GNU/Linux, Mac OS X, Windows 95 and beyond, and many others.

Exception handling. Ruby can recover from errors just like Java, C++ or Python.

The most used version of Ruby is version 1.8.7, an implementation written in C which interprets the language. This interpreter is known as MRI—*Matz Ruby Interpreter*. The most recent version of Ruby is version 1.9 where Ruby is compiled to *bytecode* before being interpreted with YARV—*Yet Another Ruby Virtual machine*—a virtual machine created for Ruby.

4.2 Ruby on Rails

Ruby on Rails, also known as Rails, is a "*framework that makes it easier to develop, deploy, and maintain web applications*" [RTH09]. It was created when David Heinemeier Hansson was working on Basecamp, an online project management service owned by 37signals, an american company which creates web applications. Ruby on Rails was part of Basecamp, from which it was extracted and released as an open source framework in July 2004. Since its creation, Rails has become a mature and popular framework, which powers well-known application such as Twitter, Hulu, iLike, Scribd, Basecamp, Shopify and Github [Rai]. The rest of this section will present the most important Rails characteristics and features for this project.

4.2.1 Philosophy

Rails' uses two very important principles that guided the Rails framework and their users, which are *convention over configuration* and *don't repeat yourself* [RTH09].

Convention over configuration seeks to decrease the number of decisions and configurations that a programmer has to do. By complying its default behavior and values, programmers only have to specify what is unconventional. Having a set of default values and procedures, Rails' programmers can enjoy its simplicity while not losing any flexibility.

Don't repeat yourself means that tany information should be located in only one place. Not only this is simpler and easier to do, both in terms of architecture and code, but

it's better prepared for the future—programmers can change something without the need to rewrite and reorganize many parts of the application, which done incorrectly may lead to inconsistency.

4.2.2 MVC

Ruby on Rails makes it easier for programmers to create applications because it imposes the usage of the *model-view-controller* (MVC) architectural pattern. This way, applications are better organized, with everything in its right place. Every Rails application is split into *models*, *views* and *controllers*.

Model. A *model* is the part responsible for maintaining the state of an application. Most times, this state is stored in a database. Besides maintaining the state of the data, it sets *constraints* on that data, e.g. a blog *Article* which must have a *title*. By keeping these constraints in the model, there's no way to break them anywhere throughout the entire application, so the data will always be consistent and valid.

View. A view generates what is displayed to users—a user interface. This is usually a visual representation of a model, but a model may have multiple views. For example, a blog will have many articles that are accessible through the model and displayed using a view. A view can have multiple formats, e.g. a RSS feed, and is used exclusively for output. Although a view may provide users with ways to input data, it never handles that same data.

Controller. The controller *controls* the entire application. It handles events from users, interacts with the model, and renders a view, as depicted in Figure 4.1. In Rails, requests pass through a routing component before reaching the controller. This component selects the appropriate controller based on the requested URL.

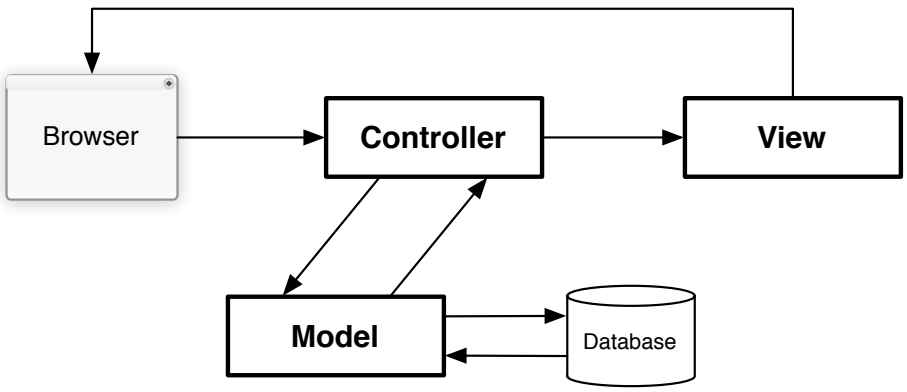

Figure 4.1: Model-View-Controller

4.2.3 ORM

Object-relational mapping is a programming technique used for abstracting database access. ORM libraries map database tables to classes. This way, programmers can use databases as though they were using simple classes, without worrying about data type conversions and database queries. A table is mapped to a class, a row is mapped to an object, and a column is mapped to objects' attributes [Amb]. So, if a programmer wants to change the name of the first user named *John*, he may do

```
johns = User.all(:name => "John")
john = johns.first
john.name = "Johnny"
john.save
```

The first and last lines of code are mapped from Ruby code to a database query. In the first line, the ORM maps the *selected* table rows to `johns`, an array of `User` objects. The last line *updates* the name of the first user to "Johnny".

Ruby on Rails includes its own ORM library, called ActiveRecord [RTH09], which also adds support for table constraints, triggers, relationships, inheritance and much more so that the programmer doesn't have to bother with complex database queries.

The first line of code is also an example of the philosophy of Rails. When the database schema file is created, the above code sample works even if the User model is defined as

```
class User < ActiveRecord::Base
end
```

One of the many Rails *conventions* is that database table names are in the plural form of model names. The model class `User` works because the table is named `users`. Because Rails *doesn't repeat itself*, the `User` model knows that a `name` field exists because that's already defined the database schema file.

Rails and ActiveRecord are capable of working regardless of the database system in use if a database adapter is available. Available adapters include MySQL, SQLite, PostgreSQL and Oracle.

4.2.4 REST

REST is an architectural style which considers the web as a collection of *resources* addressable with *Uniform Resource Identifiers* (URIs). A resource is the conceptual target of its URI. Each resource may have multiple representations, and provides a common interface to act upon it [FT00]. To make this clearer, an webpage isn't a resource, it's a representation of a resource in HTML, normally accessed through a GET request on its URI [Tom04]. However, there are other methods such and GET, which are used in REST

to use resources as listed in Table 4.1. Notice the similarity between a method's name and the *verbs* used in the description of the result action.

Table 4.1: HTTP methods used in REST

METHOD	ACTION
GET	*gets* information from the resource
POST	*adds* information to the resource
UPDATE	*updates* information in the resource
DELETE	*deletes* the resource

Rails can take advantage of REST principles to use shorter and clearer URLs and provide programmers with an easy way to create APIs in many formats. Taking a blog for example, REST can be used in Rails to create an interface such as the one listed by Table 4.2. It's important to notice that Rails can also render non-HTML views, provided that the developer implements the necessary requirements.

Table 4.2: REST on Rails

URL	METHOD	ACTION
`/articles`	GET	displays a list of articles
`/articles.rss`	GET	displays a list of articles in the RSS format
`/articles`	POST	adds a new article to the article list, passed on the request body
`/articles/1`	GET	displays a specific article
`/articles/1.xml`	GET	displays a specific article in the XML format
`/articles/1.json`	PUT	updates a specific article with the data passed on the request body as a JSON object
`/articles/1`	DELETE	deletes a specific article

4.2.5 Plugins

Other benefit of using Rails its vast collection of user-developed plugins. As of today, there are over 1000 plugins [Cur] that extend the Rails core, plugins that provide multiple authentication methods, plugins that enhance the views, plugins for multimedia, file management, testing, searching, web services, optimization, security, integration, and many more. And because Rails is Ruby, it can also use user-made Ruby libraries too. This effectively helps to make the development process faster, as it's easy to find libraries and plugins with very high quality and tested by both developers and users.

4.2.6 Data formats

Content Blaster supports third-party applications with its REST API. Nowadays, most applications' APIs are available in many data formats, being XML and JSON the most

popular ones. Both formats are used to serialize data, and Rails has built-in support for conversion between ActiveRecord and XML/JSON. This section will take a closer look at these formats.

XML

XML is a text-based general-purpose markup language. It's considered an extensible language, because it allows users to create custom markup languages. It's fundamentally important on the Web, where it's used for data exchange [W3C]. In the context of Ruby on Rails, it's used for providing and consuming web services. Take for example this code sample:

```
<?xml version="1.0" encoding="UTF-8"?>
<user>
  <id type="integer">1</id>
  <name>John</name>
  <age type="integer">34</age>
</user>
```

This code is very easy to read, and represents a user named "John", aged 34 and whose *id* is 1. This could be the response for a request on the path `/users/1.xml`, as it represents an ActiveRecord object serialized into XML. As XML is supported by most programming languages, it's a common format for an APIs that want to broaden the potential set of third-party applications using its services.

JSON

JSON is a data interchange format, based on a subset of JavaScript, a popular client-side scripting language. It's a text-based very lightweight alternative to XML for data exchange [JSOa]. Take the following code as an example:

```
{"user": {"id": 1, "name": "John", "age": 34}}
```

The code represents a JSON object, and is equivalent to the XML code of 4.2.6. Compared to XML, JSON is simpler and more readable. Unlike XML, JSON doesn't need to specify the data type of some attributes—being a subset of JavaScript, a value can hold strings, numbers, booleans, null, arrays or other JSON objects—but it isn't extensible, so it's limited to these types [JSOb].

JSON is rising in popularity for simple data exchange when extensibility isn't required. JSON code is easier to read and write, much smaller and faster to parse.

4.2.7 Summary

Ruby on Rails' vast set of features helps developers to focus and work on their problem instead of laying groundwork for their applications. By complying to Rails conventions and taking advantage of both Rails built-in features as well as third-party libraries and plugins, developers can get their work done much faster. Rails has its downsides too, being the main ones the learning curve and application performance, due in part to Ruby's performance. While the first one is rendered irrelevant due to the author's past experience with the framework, performance can be improved because Rails is flexible enough to skip its own processes when needed and to cooperate with other languages and applications—if needed, a developer may create Ruby libraries using C, which can be used by Rails as if they were pure Ruby libraries. It's important to notice that, while scaling is more of a problem in Ruby on Rails than in other frameworks and languages, very popular websites such as Twitter, Hulu, Basecamp and Shopify are developed in Ruby on Rails, which is an evidence that Rails is capable of scaling if optimized properly.

4.3 MySQL

MySQL is an open source relational database management system with over 100 million copies distributed throughout the world, with customers including many of the world's most popular websites, such as Wikipedia, Facebook or Digg. It's provided free of charge, but there's an enterprise version that provides support to its customers [Suna, Sunb]. MySQL was originally developed by a swedish company, but it was bought by Sun Microsystems in Februrary 2008 for 1000 million dollars [Sun08]. It's a fast system, easy to setup, contains a vast documentation and a large developer community, works perfectly with Rails and provides everything needed for Content Blaster to work.

4.4 Apache HTTP Server

The Apache HTTP Server is a robust, full-featured and open source HTTP server developed by the Apache Software Foundation. It provides a vast set of important features, such as basic authentication, digest authentication, content negotiation, HTTPS, virtual hosting, CGI and IPv6. It's available for most operating systems, such as GNU/Linux, FreeBSD, Solaris, Mac OS X and Microsoft Windows [Apa].

Because it can be extended with modules, Apache supports popular programming languages for web development such as PHP, Python on Perl. Phusion Passenger, also known as mod_rails, is an *open source* module for the Apache HTTP Server for deployment of Ruby and Ruby on Rails applications, created by Phusion [Phu]. As it's the easiest way

to deploy a Rails application, it has become the *de facto* standard in the Rails community for production deployment, and Phusion also provides commercial support.

4.5 Chapter summary

This chapter presented the technologies used by Content Blaster, starting by Ruby on Rails, a full-featured framework that provides a set of conventions, features—such as JSON/XML support, MVC and ORM—and plugins that make it very easy for developers to create and maintain web applications rapidly.

Ruby on Rails is built on top of Ruby, a multi-purpose dynamic and powerful language that can also be extended with C-based libraries when extreme performance is required.

The natural companions of Ruby on Rails are MySQL, Apache and Passenger, robust *open source* tools that work perfectly with Rails, a setup that has become the standard for developing Ruby on Rails applications.

Chapter 5

Design

This chapter will cover Content Blaster's design, providing detail in both logical and physical architecture, and highlighting the main design decisions taken during the project.

5.1 Logical Architecture

Logical architecture focuses on the system's organization in terms of components and entities. This section covers these topics, explaining the division of Content Blaster into four components, how they relate, and which entities are used along the entire application.

5.1.1 Components

Content Blaster is divided into four main components, *Database*, *Slope One*, *Tagging* and *Application* as depicted in Figure 5.1.

Database. The database component is a MySQL database designed to map the system's entities and their relations. It's the central repository for all the data used by the three other components.

Slope One. Slope One is the item-based collaborative filtering algorithm that powers Content Blaster. Its characteristics enforced the separation in two phases. In the first one, the *deviations* phase, the algorithm computes item-to-item deviations between every pair of items rated by at least one common user. Next follows the *predictions* phase, which uses both the users' ratings and their items' deviations. This step generates predictions for all users, attributing a predicted value to unrated user-item pairs. These steps are normally run sequentially, but the current design allows one step to be independently executed.

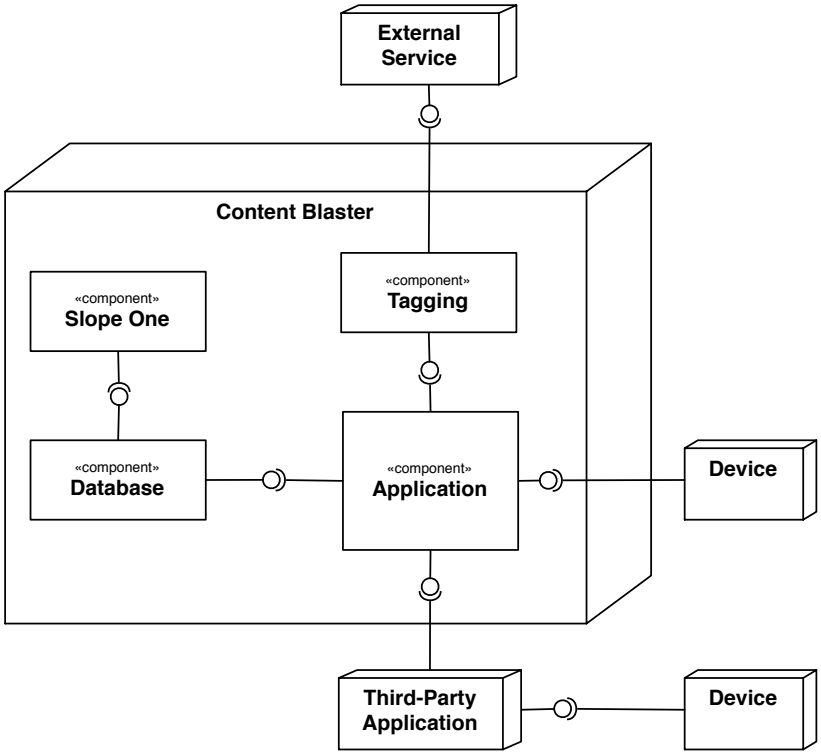

Figure 5.1: Content Blaster Component Diagram

Tagging. The tagging module extracts tags from third-party *tag providers*. Its current design consists of a class for each supported provider, which manages all the data interchange. All of these classes provide a common interface, returning a list of tags when passed a valid URL. The module also features a main class, which uses these interfaces to collect and merge the *tags* provided by each *tag provider*. This design makes it easy to add, change or remove services without affecting the other ones.

Application. This component is the core of Content Blaster and is responsible for holding the business logic and keeping its integrity, security and safety; enforcing constraints and triggers on entities and their relations; maintaining the system's state stored in the database; using the tagging module when a new item enters the system; and interacting with both users and third-party applications using the API.

5.1.2 Entities

Both the Slope One, application and tagging components use the system-wide defined entity set. This section provides an overview on these entities and their relationships, under the form of the class diagram of Figure 5.2 and a description of each one of these entities and relationships.

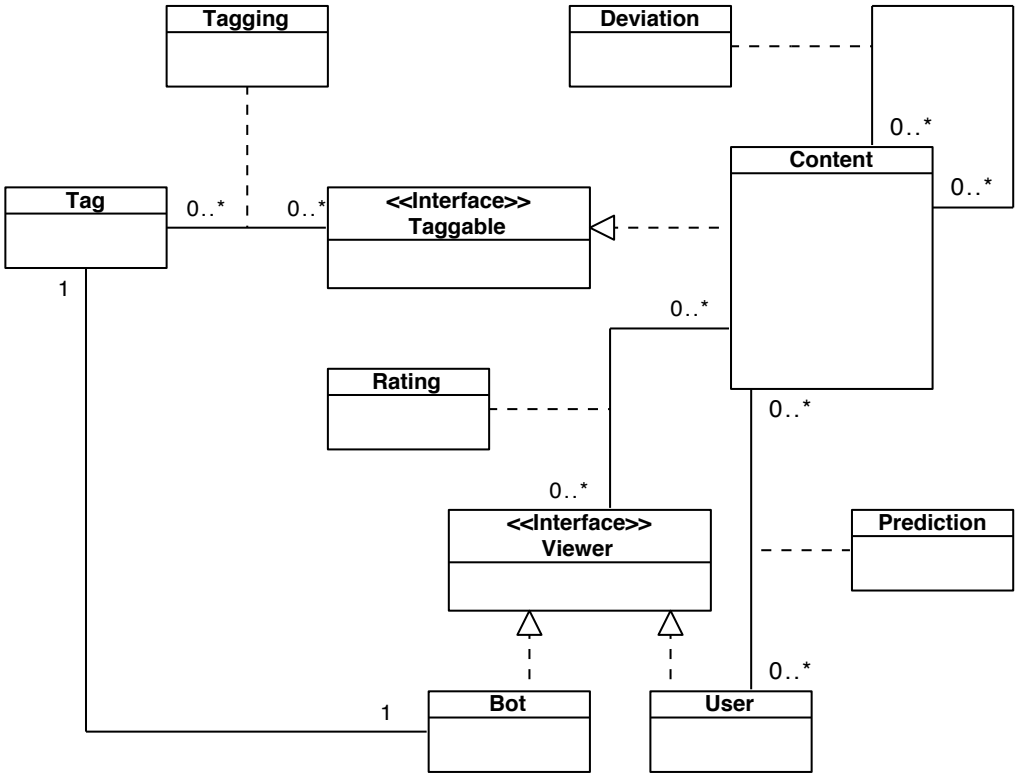

Figure 5.2: Content Blaster Class Diagram

Content. The reason why Content Blaster exists is to recommend *content*—known as *items* to the recommender system. A *Content* has an unique URL and may have multiple *tags*.

Viewer. Content Blaster needs ratings to provide recommendations to its users. An individual who rates a *Content* is known as a *Viewer*. This entity is just an interface entity, because both *users* and *filterbots* can rate *Content*.

Rating. The *Rating* entity holds the relationship between a *Content* and a *Viewer*, quantified by a numeric value: 10 if the viewer likes the content, and 0 if he doesn't.

User. An user is an human individual who rates content and to whom recommendations are given. An user must provide an username, password and email when registering, with the username and password being used for authentication. The system automatically generates a security token that is used for API authentication.

Bot. This entity represents the *filterbot* that automatically rates Content based on its tags.

Tag. Both users and the tagging module may provide *tags* to describe *content*. *Tag* is the entity which represents these tags, and it holds a single word. A *Content* is a *Taggable* entity which may have multiple tags through the *Tagging* entity, which represents the relationship between a *Tag* and a *Content*. When a new tag is created,

the system automatically creates a *Bot* associated with it. When a content is tagged, it's automatically liked by the *Bot* associated with that tag.

Deviation. This entity holds the relationship between two *contents* in the form of a numeric value computed by Slope One, noted as $dev_{i,j}$. As this value is fundamental to generate predictions, the system needs to be able to differentiate between item i and j, given $dev_{i,j} \neq dev_{j,i}$. The *Deviation* entity holds i as the *pivot* element, and j as the *deviant* element.

Prediction. A *Prediction* holds the relationship between a *Content* and a *User*, quantified with a numeric value computed by the Slope One algorithm.

5.2 Physical Architecture

Physical architecture focuses on the networking layering that Content Blast is part of. Figure 5.3 represents the simple physical architecture of Content Blaster. It runs on a single server, where users and API consumers connect to it over the Internet using the HTTP protocol. Content Blaster also connects to multiple *tag providers* using HTTP.

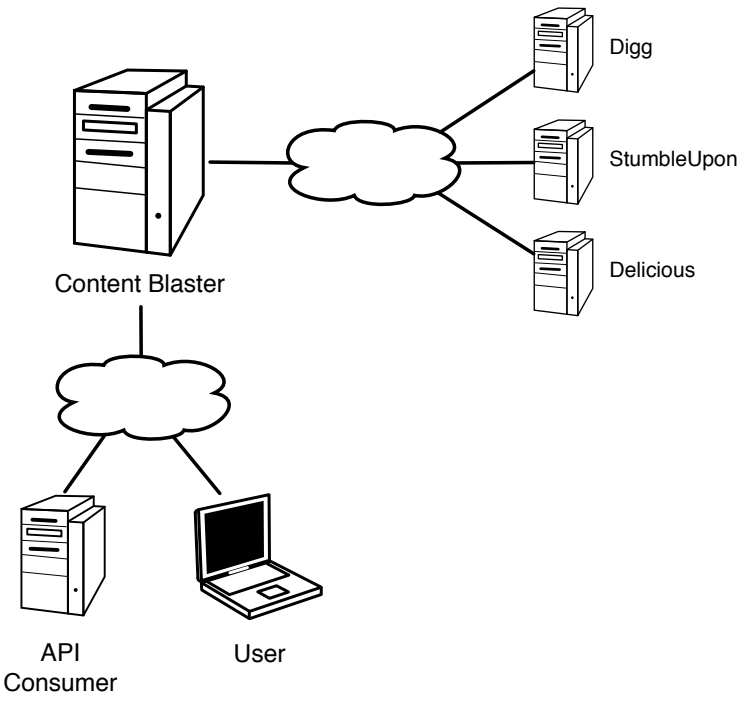

Figure 5.3: Content Blaster Network Diagram

5.3 Decisions

This section focuses on the major design decisions taken during development.

5.3.1 MVC Application

The most important decision in terms of design still left to explain is the usage of a MVC architecture in the application module. Figure 5.4 represents the models, views and controllers of the system. The application has more models than those which are depicted in the figure—one model for each entity listed in 5.1.2—but the diagram only represents the ones that interact with the controllers available for end-users and their most important views. A description of each controller and its responsibilities follows.

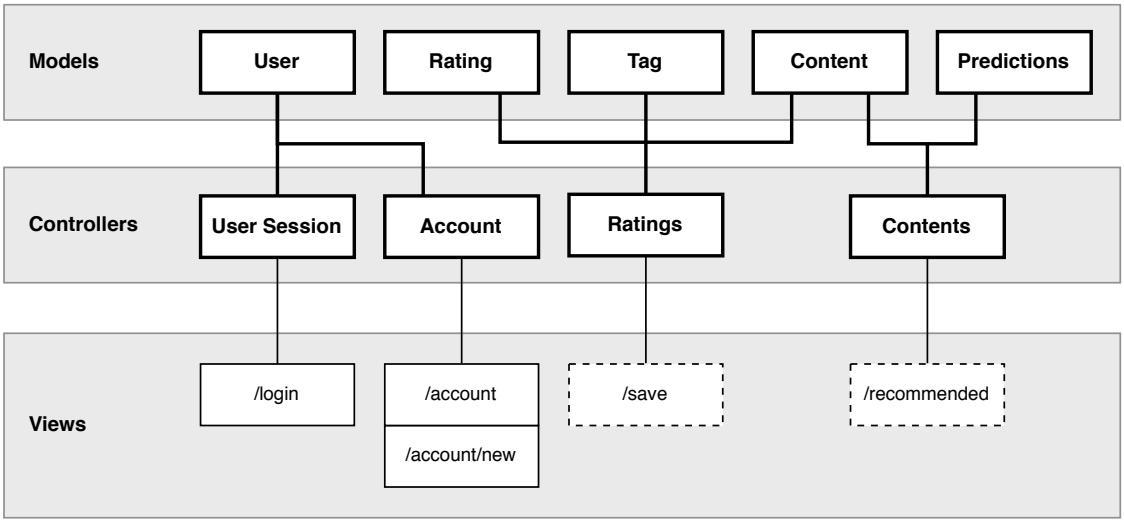

Figure 5.4: Application MVC Diagram

User Session. Handles users' session management, providing *login* and *logout*. Displays the login form.

Account. Creates user accounts, displays the account creation form and the account information screen for the current user.

Ratings. Manages the users' rating process, displaying the rating form when the user clicks the "Like/Dislike" bookmarklet button.

Contents. Displays recommendations to users. When a user clicks "Blast Me!" this controller renders a view which informs the user that he is being redirected to the recommended content.

5.3.2 API Endpoints

Another important decision to make was the API design—how to handle authentication, data formats and REST. The Content Blaster API is completely RESTful, available in both XML and JSON, and is part of the same MVC application presented before, but it only uses the Ratings and Contents controllers. To handle authentication, the user must pass his security token as a request parameter. Table 5.1 lists the available endpoints of Content Blaster's API in the JSON format.

Table 5.1: API Endpoints

METHOD	URL	ACTION
POST	`/ratings.json`	Rates an item
GET	`/contents/recommended.json`	Returns a list of recommended items

For the XML format, URLs must end with `.xml` instead of `.json`.

5.4 Chapter summary

This chapter provided an overview on Content Blaster's design. It's divided into four modules: a *Database* which stores all the data, *Slope One*, which implements the recommendation algorithm; *Tagging*, which interacts with tag providers for the system's contents; and *Application*, the core of Content Blaster, which is responsible for holding the business logic together, interacting with users and using the other modules. The system also has multiple entities, such as *users*, *bots*, *contents* and *tags*, as well as entities expressing relations between them, such as *ratings*, *deviations*, *taggings* and *predictions*.

Content Blaster is accessed over the Internet using HTTP, but it also uses third-party services over the same infrastructure.

Finally, the section explained how the application module is designed using the Model-View-Controller architectural pattern and presented the API provided by Content Blaster, which is REST-compliant and available in both XML and JSON formats.

Chapter 6

Implementation

This section covers the most important implementation details on each of Content Blaster's main components.

6.1 Database

The database is mapped from system's entities and relations, as depicted in Figure 6.1.

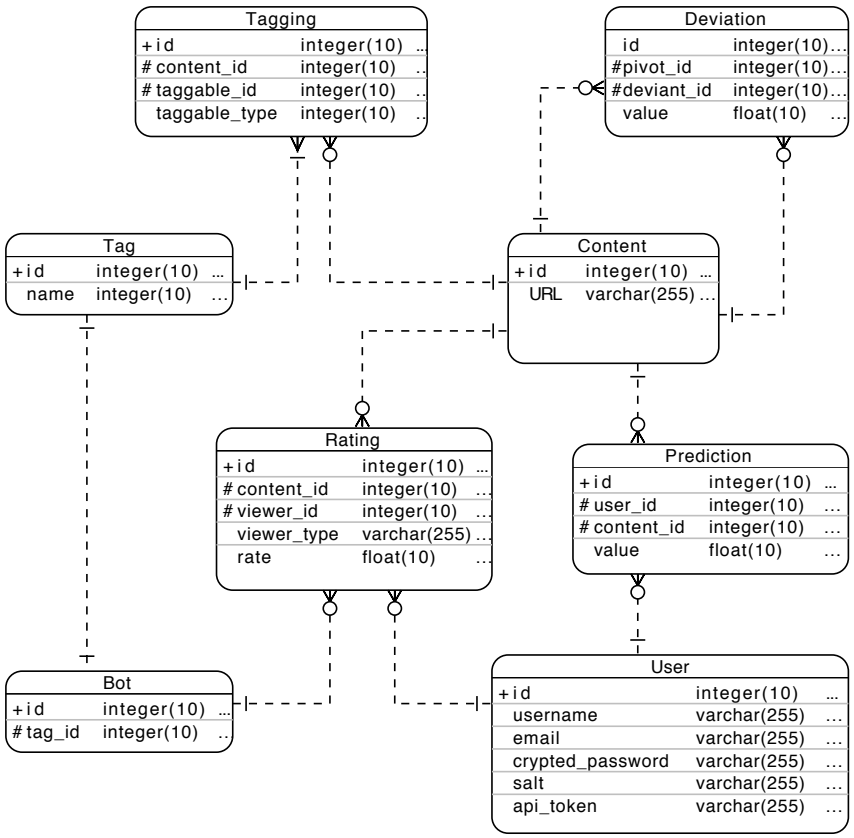

Figure 6.1: Database Entity Relationship Diagram

There are two important details that need some explanation.

- A tag only exists in the database if it's tagging at least one content. For example, if the only item tagged with a tag *music* loses that tag or is destroyed, the *music* tag, its associated bot and ratings are deleted as well.

- The class diagram of Figure 5.2 had two interface entities: *Viewer* and *Taggable*. The workaround used by Rails in such case is to add a field to the relationship table. This field holds the type of the realization of that interface. For example, both users and bots can have ratings, but there's no viewers table in the database—a *rating* has a *viewer_type*, which can be "user" or "bot". Rails infers which relationship to build based on this field.

6.2 Slope One

The Slope Algorithm is implemented in Ruby and SQL. All the facilities provided by Rails' ActiveRecord were avoided, as they degrade performance a little. However, for hundreds or thousands of queries, this little performance loss results into a major one. The algorithm is divided into two phases which may run independently: *computing deviations* and *generating predictions*.

Despite the fact that this algorithm could have been implemented in another language, Ruby was chosen because it integrates very well with the rest of the system and also helps to make the development faster, as it's easier to code. The implementation uses Ruby for the most part, using only some SQL queries to fetch data from the database. One of the major drawbacks of Ruby is its performance, but making calculations in SQL proved to be slower in comparison to Ruby. During development, three variants of the deviations' computation phase were implemented and compared in terms of performance. The results of this comparison are provided in Chapter 7.

This section will now present some implementation details of each phase.

6.2.1 Computing deviations

This phase computes item-to-item deviations using the Slope One algorithm. It's divided into three sequential steps:

1. Fetching ratings

2. Making calculations

3. Storing results

Fetching ratings

Deviations are calculated using the ratings of both users and *filterbots*. This step fetches all the ratings from the database using SQL, and stores them in an array. In this array, the element with index i stores all the viewers which rated the content with id i and the value of their rating. The following code snippet exemplifies such structure for contents with ids 244 and 245.

```
244 => {:User_25  => 10.0,
        :Bot_14   => 10.0},
245 => {:User_25  => 1.0,
        :User_30  => 10.0,
        :Bot_30   => 1.0}
```

Making calculations

This step calculates the item-to-item deviations defined in Equation 2.10 and it uses the previously created structure. One option to compute deviations is to iterate through every possible pair of items. However, that would take $n^2 - n$ iterations, considering n as the total number of items. Considering that deviations are symmetrical, $dev_{i,j} = -dev_{j,i}$, this step only computes deviations for the pairs where the *pivot* id is lower than its deviated items' ids, that is $\{dev_{i,j} | i \in I \wedge j \in I \wedge i < j\}$. This only requires $\frac{n^2-n}{2}$ iterations, half of the previous number.

For each item i in the previously created structure, the algorithm checks each other item j with an higher id and selects the common viewers between them. For these viewers, it calculates the deviation as the average difference between the viewer's ratings on item i and j. This is better understood by looking at the following pseudocode.

```
FOR EACH item_i
  FOR EACH item_j with higher id than item_i
    common_viewers = intersect(item_i.viewers, item_j.viewers)
    sum = 0
    FOR EACH viewer IN common_viewers
      sum += item_i[viewer].rate - item_j[viewer].rate
    deviation = sum / common_viewers.size
```

Storing results

This step executes the SQL query that saves all the computed deviations to the database, in the *deviations* table.

6.2.2 Generating predictions

This step generates predictions using the basic Slope One scheme defined in Equation 2.12. To generate predictions for an item, the scheme calculates the average deviation of that item in relation to the items already rated by the user. This means that in the deviation pair, the pivot element must *not have been* rated by the user. The Content Blaster implementation iterates through all users, generating predictions for each one. This section explains how predictions are computed for a single user. The algorithm executes the following steps:

1. Selects all the items which were rated by the user using an SQL query.
2. Uses an SQL query to fetch all the deviations that contain at least one item rated by the user. It does not check whether the rated item is the pivot or the deviant element of the deviation, because the previous phase only saved those deviations in which the pivot element id is smaller than the deviant element id.
3. Iterates through every user deviation:
   (a) Checks if the user rated the pivot element, the deviation element, or both:
      - If the user only rated the deviant the algorithm continues.
      - If the user only rated the pivot, the algorithm uses the reverse deviation $dev_{deviant,pivot} = -dev_{pivot,deviant}$. This is done because the Slope One scheme only considers the deviations where the pivot element wasn't already rated by the user, but the previous phase only saved those deviations where the pivot element id is smaller than the deviant element id. For this deviation, it considers the deviant element as the pivot element and vice-versa, using the symmetrical value.
      - If the user has rated both, the algorithm skips this iteration.

   (b) Saves the deviation value to an *accumulator* variable. This *accumulator* sums all the deviations' values, grouped by their pivot element, and also stores how many values where already summed for that pivot element. Using the sum and the total of values summed, the mean value can be calculated later.
4. For each pivot in the accumulator, it sums the user's average rating with the mean value of that pivot's deviations, which gives the predicted value for that pivot.
5. Stores all the user's predictions in the *predictions* table on the database.

6.3 Tagging

The tagging component consists of a central module which interacts with the modules responsible for each supported third-party service. These modules are built in Ruby, and

they must fulfill two requirements. First, they must have an `url_tags` method, which accepts an URL string. Second, that method must return a predefined structure representing the retrieved tags and format for that URL. Because some services allow their users to have their own tags, it also includes the number of occurrences of each tag. Each service may use whatever technique or implementation is desired, as long as it follows these rules. An example of the structure returned by a service may be

```
{:format => {:video    => 1},
 :tags   => {"amelie"  => 25,
             "trailer" => 25,
             "movie"   => 21,
             "french"  => 2}}
```

As mentioned before, the services currently supported by Content Blaster are Digg, StumbleUpon and Delicious. All of these services use the HTTParty library for consuming web services, with automatic parsing of JSON and XML responses [Nun08].

While Digg and Delicious both JSON APIs which make it very easy to retrieve the tags, StumbleUpon doesn't have an API. Instead, Content Blaster gets the HTML of StumbleUpon's page on that URL. These pages display information about the page, including the tags, so Content Blaster needs to parse those pages to retrieve the tags.

6.4 Application

The application component, as the core component that interacts with users and controls the other components, has a reasonable level of complexity. Because Content Blaster follows Rails' conventions, most of it is easily handled with the framework. Even the authentication method, which is usually a complex feature, was easily done using the Authlogic plugin [Bin08]. This plugin automatically generates a secure encrypted password when users register an account, creates a security token used for API authentication, enhances security with protection from brute-force techniques and provides several useful methods to manage access control in both the interface and the API.

However, there are two important implementation details that require further explanation: the technique used for creating *bookmarklets*, and the format for *API* calls and responses.

6.4.1 Bookmarklets

Bookmarklets are regular bookmarks that execute some JavaScript code when clicked, instead of opening a new page, and are supported in every major web browser. JavaScript is a popular client-side scripting language which is also supported by every major browser.

Bookmarklets are used in two different cases, *rating a page* and *getting a recommendation*.

Rating a page. The bookmarklet for the "Like/Dislike" button is fairly simple. It uses JavaScript methods to open a new popup window with a custom size, which displays a view from the *Ratings* controller with a form that allows users to save the rating.

Getting a recommendation. This bookmarklet changes the URL in the browser's address bar to a view from the *Contents* controller, which then redirects the user to a recommendation.

6.4.2 API calls and response

As mentioned before in Section 5.3.2, the Content Blaster API provides two endpoints for third-party developers, one to rate items with a POST request, and the other to get a list of recommendations with a GET request. This section cover the specification of the API, detailing the required request parameters and the response content and status codes for both errors and successful calls.

Response codes

Content Blaster API returns appropriate HTTP status codes in every request:

200 OK Successful call.

302 Not Modified No new data to return, used when there are no recommendations to provide.

400 Bad Request Request is invalid, perhaps required parameters are missing.

403 Forbidden Wrong API token.

404 Not Found Invalid URI requested.

500 Internal Server Error Internal malfunction in Content Blaster.

Error messages

Content Blaster API returns error messages in both JSON and XML formats, explaining what is wrong and what the user can do to overcome the problem. A possible error in XML could be this one:

```
<?xml version="1.0" encoding="UTF-8"?>
<hash>
  <request>/contents/recommended</request>
```

```
  <error>You do not have privileges to access this page.</error>
</hash>
```

In JSON, the same error looks like this:

```
{ "request": "/contents/recommended",
  "error": "You do not have privileges to access this page."}
```

Authentication

Endpoints which require authentication need a `api_key` parameter in the URL. This parameter is the user's security token which can be found in Content Blaster's homepage when authenticated.

Recommendations

This API endpoint provides a list of 10 recommendations.

Relative path
`/contents/recommended.{xml|json}`

Method
GET

Requires authentication
Yes.

Parameters
None required.

Response
This sample response lists only 2 recommendations instead of the usual 10.

```
<?xml version="1.0" encoding="UTF-8"?>
<contents type="array">
  <content>
    <url>http://www.stickyscreen.org/</url>
    <created-at>2008-10-25T21:44:13Z</created-at>
  </content>
  <content>
    <url>http://justwatchthesky.com/</url>
    <created-at>2008-12-30T15:05:47Z</created-at>
```

```
  </content>
</contents>
```

Rating

This API endpoint allows a user to rate a content.

Relative path

`/ratings.{xml|json}`

Method

POST

Requires authentication

Yes.

Parameters

The parameters for this endpoint include containers that contain other parameters.

- `rating`
  - `value`. Required. Must be either `like` or `dislike`.
  - `content`.
    - `url`. Required. Must be a valid URL.
    - `tags`. Optional. List of space-separated words.

Response

The response is the rated content.

```
{ "content":
  { "url": "http://www.stickyscreen.org/",
    "created_at": "2008-10-25T21:44:13Z" }}
```

Sample request body

This is a sample *like* request with three tags.

```
<rating>
  <value>like</value>
  <content>
    <url>http://justwatchthesky.com/</url>
```

```
    <tags>typography design music</tags>
  </content>
</rating>
```

6.5 Chapter summary

This chapter covered some implementation details of the project, starting by the database and the recommender algorithm. This algorithm is executed mostly in Ruby and fetches data with pure SQL queries instead of ActiveRecord, as it degrades performance. An important optimization was to compute only half of the item-to-item deviations, as the deviations matrix is symmetrical. Next, the section covered some details of the application, including the *bookmarklets* technique and the API calls and responses. The Content Blaster API is completely RESTful, supports XML and JSON, and makes appropriate use of HTTP status codes.

Chapter 7

Solution Evaluation

Usually, recommender systems are evaluated in terms of performance and accuracy. In terms of accuracy, recommender systems use samples of very large publicly available data sets, measuring the error of their computed predictions. Available data sets include the MovieLens, EachMovie, Book-crossing and Jester Jokes data sets from the GroupLens research group [Gro].

However, there's no publicly and legally available data set for websites, and Content Blaster is a recent project, which current data set contains approximately 100 items, 12 users and 200 user ratings. For comparison, [SKKR01] uses a test data set of approximately 950 users, 1600 items and 100000 ratings. It's impossible to make a reasonable accuracy analysis using Content Blaster's tiny data set. Additionally, as the algorithm used by Content Blaster was already subject to comparison with other algorithms in terms of accuracy, measuring it isn't very fundamental in this project either.

What is fundamental to analyze and discuss is the performance of the algorithm, being this a Ruby web-based recommender system. This section covers the performance analysis of the algorithm used in Content Blaster, comparing it to other implementations developed during the project, and withdrawing conclusions from it.

7.1 Algorithms

It was decided to compare three different implementations of the Slope One algorithm:

SQL. A single SQL query which computes all the deviations.

SQL Batches. Multiple SQL queries, each of them computing deviations for an item.

Ruby. Using SQL only to fetch data, calculations are done in Ruby.

Before they were tested, it was expected that the *SQL* implementation would be the fastest, as Ruby is reportedly slow. *SQL Batches* is very similar to the first one, but was developed

to alleviate the database from extreme load during a long period, which should be caused by the *SQL* implementation. The *Ruby* implementation was expected to be the slower, as it uses Ruby for the most part, in contrast to the first one. The *SQL* and *SQL Batches* queries can be found in Appendixes E and F, respectively.

7.2 Methodology and results

The three versions were implemented and compared in terms of performance, using a synthesized data set with 400 users, 1000 items and 80000 ratings. The versions were tested in order (*SQL*, *SQL batches* and *Ruby*), computing and inserting the deviations in the database, with the deviations table being cleared in between. Each version was tested five times. The run times, in seconds, are listed in Table 7.1.

Table 7.1: Run times of the algorithm implementations

	t_1	t_2	t_3	t_4	t_5	$\bar{t}$
SQL	91	90	91	91	91	**90,8**
SQL BATCHES	93	94	93	95	92	**93,4**
RUBY	78	79	79	78	78	**78,4**

Before diving into what can be concluded from the test, it's important to refer how the run times for the *SQL* and *SQL Batches* implementations were achieved. With the default MySQL configuration, the best time for the *SQL* implementation was over 250 seconds. As these results were totally unacceptable, a proper MySQL configuration was deemed necessary. These results use the best of *ten* different MySQL configuration files.

Even more important is the fact that the performance of these SQL-based algorithms is *very sensitive* to the MySQL configuration. The process of optimizing this configuration for optimal results was anything but straightforward, as it isn't very clear how some parameters affect the run times. Sometimes, making more memory available for MySQL resulted in improved performance in one implementation, but worse in the other. The results listed in Table 7.1 use the best overall configuration in the tests that were performed. This configuration can be consulted on Appendix G.

The MySQL ratings table had was also optimized, with the necessary indexes for optimized query performance.

7.3 Discussion

The results were surprising, as the *Ruby* implementation proved better every time than the SQL-based implementations, contrary the expectations. This is specially relevant as the SQL-based implementations were optimized during the course of days, and the ratings table had the necessary indexes for the queries performed. The *Ruby* implementation

results were the best overall with an average of 78,4 seconds, which is 86% of the next best implementation with 90,8 seconds. The *SQL Batches* implementation proved to be the worst implementation, with an average of 93,4 seconds, and with unstable results, as low as 92 seconds but peeking at 95. Both the *Ruby* and the *SQL* implementation were very stable.

While the SQL-based implementations were subject to configuration and indexes optimizations, the *Ruby* implementation has further room for improvement. First, it's possible to use the JRuby—a Ruby interpreter based on Java—which adds support for native operating system threads, making Ruby capable of using multiple processors, contrary to the standard implementation. One other option is to make a Ruby extension in C, which may dramatically improve performance, as C is many times faster than Ruby, or using a complete different language.

Overall, it's best to rely on MySQL as little as possible. Besides being very sensitive to configuration, programming languages may offer other capabilities, such as native threading and distributed computing. Other factor to consider is that putting the database in an extreme load affects the users, as they're constantly using the database when interacting with Content Blaster.

Considering these results and what has been discussed, the other phase of the algorithm—generating the predictions—followed the same ideas, using Ruby for the most part and SQL exclusively for fetching the necessary data.

7.4 Chapter summary

This chapter covered the performance analysis of three different implementations to *compute deviations* in the Slope One algorithm. One used a single SQL query to make all the calculations; other used one SQL query for each item, which did all the necessary calculations for that item; and the last used Ruby for the most part, using SQL only to fetch the data from the database. Tests proved that Ruby provided the best overall performance, which could be further improved with native multi-threading or using another programming language.

It also explained why it wasn't considered fundamental to analyze the accuracy of recommendations, as Slope One was already tested against other algorithms, and the current Content Blaster data set is too small to make a feasible analysis.

Chapter 8

Conclusions and Future Work

This section presents the conclusions gathered from this project and what may be done to make Content Blaster better in the future.

8.1 Conclusions

The main objectives presented in Section 1.3 have been fulfilled. Content Blaster is a web-based recommendation system that makes it very easy for a user to express their preference over any webpage their visiting.

It works with two very basic buttons on the browser, and it can even be used on a last-generation smartphone such as the iPhone. One of the motivations behind this work was to strive for ubiquity in recommendations, which is only possible through the available API, which is also very simple to use and makes a proper use of HTTP to provide helpful error messages to third-party developers.

During the project's development, it became clear that Ruby on Rails was a very good choice for Content Blaster. All the groundwork provided by ActiveRecord and the MVC architecture was fundamental in making the system operational very quickly. The vast array of available plugins was very helpful for authentication, HTTP requests in the tagging module, access control and HTML parsing.

The precious time saved by using Rails was then available to focus on the implementation and optimization of the Slope One algorithm. Slope One's implementation proved reasonably easy, and the performance tests were fundamental. Without them, Content Blaster would have kept using the first implemented method, which computed the item-to-item deviations in one huge SQL query. Even though the development of both SQL-based implementations was very time-expensive and they ended up not being used, they were valuable in determining which implementation to use. Making the decision to go

with Slope One was hugely beneficial for this project, and having enough information to consciously make that decision was fundamental.

What's interesting and original in Content Blaster is that it leverages the power of the users by combining tags retrieved from other websites as *filterbots* in a recommender system. This may prove to be very important, but only when the data set is large enough will it be possible to analyze its accuracy.

Content Blaster is now up, running and providing recommendations. Its components are clearly separated and organized, and each of them may be improved or rebuilt independently, combined in a simple but solid infrastructure.

8.2 Limitations

Content Blaster still has some limitations which couldn't be addressed during the course of this project.

First and foremost, there's no clear prediction on how scalable the system is. Although the algorithms were tested with large data sets, that's not the same as having to generate recommendations *and* still be a robust and responsive system to its users and third-party developers. Content Blaster's performance must be constantly monitored as the system grows, and it may be necessary to physically separate the Slope One algorithm from the rest of the application, so it can use dedicated resources.

Another current limitation is the server infrastructure. Content Blaster runs on a shared server alongside other projects, which is the main reason for computing predictions *offline* and storing them in the database.

8.3 Future Work

Content Blaster is now a solid infrastructure that may be improved. During the course of the project, many ideas were considered, but most of them couldn't have been implemented within the available timeframe. This section presents some of those ideas.

8.3.1 Content visualization

Each content recommended is displayed as is: a regular webpage. However, Content Blaster could enhance how users visualize their content. For example, videos from popular websites like YouTube of Vimeo could be extracted from their full rendered page so the user can focus on what really matters. Another idea is to improve the readability of news websites, using a bigger font size and removing a great part of the surrounding elements. For photo websites such as Flickr or Picasa, Content Blaster could display a better photo gallery, with improved navigation and full-sized images.

8.3.2 Improving recommendations

Although the system's performance has been strongly considered during this project, it could be further improve. This is also true in terms of accuracy. Possible solutions include testing the *Weighted Slope One* scheme, pre-computing which pairs of items have a common user, experimenting with other database management systems, not storing all the item-to-item deviations, using an item neighborhood to compute predictions or testing the algorithm in other programming languages.

8.3.3 New features

Currently, the application component of Content Blaster doesn't provide many features to its users. Future work may add features such as browsing all the content in the system by tags, searching or filtering recommendations based on a search query or a particular tag, and importing rated items from other websites such as Digg or StumbleUpon. This way, users could experiment with other ways of finding content, spending more time and rating more items, which effectively improves the system's data set.

8.3.4 Extended tagging

Nowadays, websites like YouTube, Flickr, any blog based on the Blogger and Wordpress engines, and news aggregators such as Technorati and Newsvine, allow their users to tag their content. The tagging system could be extended to support those websites, retrieving tags from a particular content page.

8.3.5 Clients for other devices

Taking advantage of the system's API, applications for other devices could be developed in the future. For example, it would be interesting for users to enjoy personalized content on their *set-top-boxes* within the comfort of their living rooms. As Content Blaster strives for ubiquity in recommendations, mobile devices such as the iPhone or Android-based phones could also be interesting platforms for development.

References

[Alea] Alexa Internet, Inc. Alexa top sites. http://www.alexa.com/topsites. Accessed 9th June 2009.

[Aleb] Alexa Internet, Inc. digg.com – traffic details from Alexa. http://www.alexa.com/siteinfo/digg.com. Accessed 23th June 2009.

[Alec] Alexa Internet, Inc. stumbleupon.com – traffic details from Alexa. http://www.alexa.com/siteinfo/stumbleupon.com. Accessed 23th June 2009.

[Amb] Scott W. Ambler. Mapping objects to relational databases: O/R mapping in detail. http://www.agiledata.org/essays/mappingObjects.html. Accessed 25th June 2009.

[Apa] Apache Software Foundation. Apache core features. http://httpd.apache.org/docs/2.2/mod/core.html. Accessed 26th June 2009.

[AT05] Gediminas Adomavicius and Er Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17:734–749, 2005.

[BH04] Justin Basilico and Thomas Hofmann. Unifying collaborative and content-based filtering. In *ICML '04: Proceedings of the twenty-first international conference on Machine learning*, page 9, New York, NY, USA, 2004. ACM.

[BHC98] Chumki Basu, Haym Hirsh, and William Cohen. Recommendation as classification: using social and content-based information in recommendation. In *AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, pages 714–720, Menlo Park, CA, USA, 1998. AAAI Press.

[BHK98] John S. Breese, David Heckerman, and Carl Kadie. Empirical analysis of predictive algorithms for collaborative filtering. Technical Report MSR-TR-98-12, Microsoft Research, 1998. Published by Morgan Kaufmann Publishers.

[Bin08] Binary Logic. Authlogic released! rails authentication done right. http://www.binarylogic.com/2008/10/25/authlogic-released-rails-authentication-done-right/, October 2008. Accessed 28th June 2009.

[BKR07] Shlomo Berkovsky, Tsvi Kuflik, and Francesco Ricci. Distributed collaborative filtering with domain specialization. In *RecSys '07: Proceedings of the 2007 ACM conference on Recommender systems*, pages 33–40, New York, NY, USA, 2007. ACM.

[BS97] Marko Balabanović and Yoav Shoham. Fab: content-based, collaborative recommendation. *Communications of the ACM*, 40(3):66–72, 1997.

[Bur02] Robin Burke. Hybrid recommender systems: Survey and experiments. *User Modeling and User-Adapted Interaction*, 12(4):331–370, 2002.

[Cer07] Vinton G. Cerf. An information avalanche. *Computer*, 40(1):104–105, 2007.

[CGM+99] Mark Claypool, Anuja Gokhale, Tim Miranda, Pavel Murnikov, Dmitry Netes, and Matthew Sartin. Combining content-based and collaborative filters in an online newspaper. In *In Proceedings of ACM SIGIR Workshop on Recommender Systems*, Berkeley, CA, USA, 1999. ACM.

[Coma] Compete, Inc. Site profile for digg.com (rank #18). http://siteanalytics.compete.com/digg.com/?metric=uv. Accessed 23th June 2009.

[Comb] Compete, Inc. Site profile for stumbleupon.com (rank #291). http://siteanalytics.compete.com/stumbleupon.com/?metric=uv. Accessed 23th June 2009.

[Cur] Benjamin Curtis. Ruby on rails plugins — agilewebdevelopment. http://agilewebdevelopment.com/. Accessed 25th June 2009.

[Del] Delicious. http://delicious.com. Accessed 24th June 2009.

[Diga] Digg. http://digg.com. Accessed 22th June 2009.

[Digb] Inc. Digg. How Digg works. http://digg.com/how. Accessed 23th June 2009.

[DK04] Mukund Deshpande and George Karypis. Item-based top-n recommendation algorithms. *CM Transactions on Information Systems (TOIS)*, 22(1):143–177, 2004.

[eBa07] eBay, Inc. eBay acquires StumbleUpon. http://investor.ebay.com/releasedetail.cfm?ReleaseID=246467, May 2007. Accessed 10th June 2009.

[FM08] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly Media, Inc., 2008.

[Fraa] Fraunhofer-Gesellschaft. Fraunhofer-Gesellschaft: About us. http://www.fraunhofer.de/EN/company/index.jsp. Accessed 11th June 2009.

[Frab] Fraunhofer Portugal. AICOS: About us. http://www.aicos.fraunhofer.pt/index.php?id=289. Accessed 11th June 2009.

[FT00] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 407–416, New York, NY, USA, 2000. ACM.

[Goo] Google. Corporate information — technology overview. http://www.google.com/corporate/tech.html. Accessed 10th June 2009.

[Gro] GroupLens Research. Data sets. http://www.grouplens.org/taxonomy/term/14. Accessed 28th June 2009.

[GSK+99] Nathaniel Good, J. Ben Schafer, Joseph A. Konstan, Al Borchers, Badrul Sarwar, Jon Herlocker, and John Riedl. Combining collaborative filtering with personal agents for better recommendations. In *In Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 439–446, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.

[HCZ04] Zan Huang, Hsinchun Chen, and Daniel Zeng. Applying associative retrieval techniques to alleviate the sparsity problem in collaborative filtering. *ACM Transactions on Information Systems (TOIS)*, 22(1):116–142, 2004.

[HKBR99] Jonathan L. Herlocker, Joseph A. Konstan, Al Borchers, and John Riedl. An algorithmic framework for performing collaborative filtering. In *SIGIR '99: Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 230–237, New York, NY, USA, 1999. ACM.

[HKR02] Jon Herlocker, Joseph A. Konstan, and John Riedl. An empirical analysis of design choices in neighborhood-based collaborative filtering algorithms. *Information Retrieval*, 5(4):287–310, 2002.

[HKTR04] Jonathan L. Herlocker, Joseph A. Konstan, Loren G. Terveen, and John T. Riedl. Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS)*, 22(1):5–53, 2004.

[IdGL+08] Leo Iaquinta, Marco de Gemmis, Pasquale Lops, Giovanni Semeraro, Michele Filannino, and Piero Molino. Introducing serendipity in a content-based recommender system. In *HIS '08: Proceedings of the 2008 8th International Conference on Hybrid Intelligent Systems*, pages 168–173, Washington, DC, USA, 2008. IEEE Computer Society.

[IT 09] IT Facts. 58% of americans have a mobile phone with web connectivity. http://www.itfacts.biz/58-of-americans-have-a-mobile-phone-with-web-connectivity/12965, May 2009. Accessed 12th June 2009.

[JSOa] JSON. http://www.json.org/. Accessed 26th June 2009.

[JSOb] JSON: The fat-free alternative to XML. http://www.json.org/xml.html. Accessed 26th June 2009.

[Kas] Anton Kast. Recommendation engine. http://digg.com/whitepapers/recommendationengine. Accessed 23th June 2009.

[Kir09] Marshall Kirkpatrick. StumbleUpon hits 7 million users, quietly 50 http://www.readwriteweb.com/archives/stumbleupon_hits_7_million_users.php, February 2009. Accessed 23th June 2009.

[KT00] Mei Kobayashi and Koichi Takeda. Information retrieval on the web. *ACM Computing Surveys (CSUR)*, 32(2):144–173, 2000.

[Las] Last.fm. About Last.fm. http://www.last.fm/about. Accessed 10th June 2009.

[LJB01] Gregory D. Linden, Jennifer A. Jacobi, and Eric A. Benson. Collaborative recommendations using item-to-item similarity mappings. United States Patent 6266649, July 2001.

[LM05] Daniel Lemire and Anna Maclachlan. Slope one predictors for online rating-based collaborative filtering. In *SDM '05: Proceedings of the Fifth SIAM International Conference on Data Mining*, pages 471–475. SIAM, 2005.

[LSY03] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80, January 2003.

[Mar06] Matt Marshall. Aggregate knowledge raises $5M from Kleiner, on a roll. http://venturebeat.com/2006/12/10/aggregate-knowledge-raises-5m-from-kleiner-on-a-roll, December 2006. Accessed 10th June 2009.

[Neta] Netflix, Inc. The Netflix prize rules. http://www.netflixprize.com/rules. Accessed 11th June 2009.

[Netb] Netflix, Inc. Press kit — Netflix facts. http://www.netflix.com/MediaCenter?id=5379. Accessed 11th June 2009.

[Nie99] Jakob Nielsen. *Designing Web Usability: The Practice of Simplicity*. New Riders Publishing, 1999.

[Nie08] Jakob Nielsen. User skills improving, but only slightly (Jakob Nielsen's Alertbox). http://www.useit.com/alertbox/user-skills.html, February 2008. Accessed 12th June 2009.

[NPD09] The NPD Group. The NPD Group: Despite recession, U.S. smartphone market is growing. http://www.npd.com/press/releases/press_090303.html, March 2009. Accessed 12th June 2009.

[Nun08] John Nunemaker. It's an HTTParty and everyone is invited! http://railstips.org/2008/7/29/it-s-an-httparty-and-everyone-is-invited, July 2008. Accessed 28th June 2009.

[PHLG00] David M. Pennock, Eric Horvitz, Steve Lawrence, and C. Lee Giles. Collaborative filtering by personality diagnosis: A hybrid memory and model-based approach. In *UAI '00: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 473–480, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[Phu] Phusion. Overview—Phusion Passenger™ (a.k.a mod_rails / mod_rack). http://www.modrails.com/. Accessed 26th June 2009.

[Por06] Joshua Porter. Watch and learn: How recommendation systems are redefining the web. http://www.uie.com/articles/recommendation_systems/, December 2006. Accessed 11th June 2009.

[PPM+06] Seung-Taek Park, David Pennock, Omid Madani, Nathan Good, and Dennis DeCoste. Naïve filterbots for robust cold-start recommendations. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 699–705, New York, NY, USA, 2006. ACM.

[Rai] Ruby on rails: Applications. http://rubyonrails.org/applications. Accessed 25th June 2009.

[RIS+94] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In *CSCW '94: Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 175–186, New York, NY, USA, 1994. ACM.

[Ros09] Kevin Rose. Recommendation engine rolling out this week! http://blog.digg.com/?p=127, June 2009. Accessed 23th June 2009.

[RTH09] Sam Ruby, Dave Thomas, and David Heinemeier Hansson. *Agile Web Development with Rails*. Pragmatic Programmers, third edition, March 2009.

[Rub] About Ruby. http://www.ruby-lang.org/en/about/. Acccessed 25th June 2009.

[Rya09] Ryan Junee, The YouTube Team. Zoinks! 20 hours of video uploaded every minute! http://www.youtube.com/blog?entry=on4EmafA5MA, May 2009. Accessed 8th June 2009.

[Seg07] Toby Segaran. *Programming collective intelligence*. O'Reilly, first edition, 2007.

[SKB+98] Badrul Sarwar, Joseph Konstan, Al Borchers, Jon Herlocker, Brad Miller, and John Riedl. Using filtering agents to improve prediction quality in the grouplens research collaborative filtering system. In *CSCW '98: Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 345–354, New York, NY, USA, 1998. ACM.

[SKKR01] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 285–295, New York, NY, USA, 2001. ACM.

[SM95] Upendra Shardanand and Pattie Maes. Social information filtering: algorithms for automating "word of mouth". In *CHI '95: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 210–217, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[Stua] StumbleUpon. http://www.stumbleupon.com/. Accessed 22th June 2009.

[Stub] StumbleUpon. About StumbleUpon. http://www.stumbleupon.com/aboutus/. Accessed 10th June 2009.

[Stuc] StumbleUpon. StumbleUpon's recommendation technology. http://www.stumbleupon.com/technology/. Accessed 12th June 2009.

[Suna] Sun Microsystems. About MySQL. http://www.mysql.com/about/. Accessed 26th June 2009.

[Sunb] Sun Microsystems. MySQL customers. http://www.mysql.com/customers/?id=34. Accessed 26th June 2009.

[Sun08] Sun Microsystems. Sun Microsystems announces completion of MySQL acquisition; paves way for secure, open source platform to power the network economy. http://www.sun.com/aboutsun/pr/2008-02/sunflash.20080226.1.xml, February 2008. Accessed 26th June 2009.

[Tim07] The New York Times. CBS buys Last.fm, an online radio site. http://www.nytimes.com/2007/05/31/business/media/31radios.html, May 2007. Accessed 10th June 2009.

[Tom04] Ryan Tomayko. How i explained REST to my wife. http://tomayko.com/writings/rest-to-my-wife, December 2004. Accessed 25th June 2009.

[USA09] USA Today. Some PC makers don't know what to do with netbooks. http://www.usatoday.com/tech/products/2009-01-19-netbooks-future_N.htm?csp=34, January 2009. Accessed 12th June 2009.

[W3C] W3C. Extensible markup language (XML). http://www.w3.org/XML/. Accessed 26th June 2009.

[Yah] Yahoo! Inc. About Delicious. http://delicious.com/about. Accessed 24th June 2009.

[ZP07] Jiyong Zhang and Pearl Pu. A recursive prediction algorithm for collaborative filtering recommender systems. In *RecSys '07: Proceedings of the 2007 ACM conference on Recommender systems*, pages 57–64, New York, NY, USA, 2007. ACM.

Appendix A

Alexa Top Sites

The top 10 websites listed in the Alexa Traffic Rank as of June 9th 2009 [Alea]:

1. **Google**, http://google.com
   Enables users to search the Web, Usenet, and images. Features include PageRank, caching and translation of results, and an option to find similar pages. The company's focus is developing search technology.

2. **Yahoo!**, http://yahoo.com
   Personalized content and search options. Chatrooms, free e-mail, clubs, and pager.

3. **YouTube**, http://youtube.com
   YouTube is a way to get your videos to the people who matter to you. Upload, tag and share your videos worldwide!

4. **Facebook**, http://facebook.com
   A social utility that connects people, to keep up with friends, upload photos, share links and videos.

5. **Windows Live**, http://live.com
   Search engine from Microsoft.

6. **Microsoft Network (MSN)**, http://msn.com
   Dialup access and content provider.

7. **Wikipedia**, http://wikipedia.org
   An online collaborative encyclopedia.

8. **Blogger.com**, http://blogger.com
   Free, automated weblog publishing tool that sends updates to a site via FTP.

9. **Baidu.com**, http://baidu.com
   The leading Chinese language search engine, provides simple and reliable search experience, strong in Chinese language and multi-media content including MP3 music and movies, the first to offer WAP and PDA-based mobile search in China.

10. **Yahoo! Japan**, http://yahoo.co.jp
    Japanese version of Yahoo!

Appendix B

StumbleUpon categories

In StumbleUpon, every category collects a large number of topics, with over 500 topics across all categories. As this number is very large, this Appendix only lists the categories, in alphabetic order:

- Arts/History
- Commerce
- Computers
- Health
- Hobbies
- Home/Living
- Media
- Music/Movies
- Outdoors
- Regional
- Religion
- Sci/Tech
- Society
- Sports

StumbleUpon categories

Appendix C

Digg topics

The topics on Digg are grouped into containers. Containers and its topics are, in the same order of Digg's frontpage:

- Technology
  - Apple
  - Design
  - Gadgets
  - Hardware
  - Tech Industry News
  - Linux/Unix
  - Microsoft
  - Mods
  - Programming
  - Security
  - Software
- World & Business
  - Business & Finance
  - World News
  - Political News
  - Political Opinion
- Entertainment
  - Celebrity
  - Movies
  - Music
  - Television
  - Comics & Animation
- Gaming

  - Gaming Industry News
  - PC Games
  - Playable Web Games
  - Nintendo
  - PlayStation
  - Xbox
- Science
  - Environment
  - General Sciences
  - Space
- Sports
  - Baseball
  - Basketball
  - Extreme
  - American & Canadian Football
  - Golf
  - Hockey
  - Motorsport
  - Olympics
  - Soccer
  - Tennis
  - Other Sports
- Lifestyle
  - Arts & Culture
  - Autos
  - Educational
  - Food & Drink
  - Health
  - Travel & Places
- Offbeat
  - Comedy
  - Odd Stuff
  - People
  - Pets & Animals

Appendix D

Transcript of conversation

This is an adapted transcript of a conversation with the creator of the Slope One algorithm, Daniel Lemire, on March 5th, 2009.

Paulo Pereira Do you know of any work addressing cold-start systems using Slope One?

Daniel Lemire I don't think that Slope One is different with relation to cold start. You can use generic cold-start strategies.

Paulo Pereira Someone could have worked on an unified approach using Slope One. I'm working on a "StumbleUpon for everything".

Daniel Lemire StumbleUpon for everything? Don't URIs cover everything already?

Paulo Pereira It's my final 5-year MsC project. Also, could Slope One work with thumbs up/down, like a discrete 0-1 scale? That was the idea.

Daniel Lemire For binary data, you might be better off with Greg Linden's item-to-item scheme.

Paulo Pereira The problem is that not only the users can "like" items, but they can also "dislike", contrary to Amazon.

Daniel Lemire Then it is not binary. And then Slope One should be applicable.

Paulo Pereira Yes. I'll use a positive number "thumbs up", but I must use zero or negative for "Thumbs DOWN". Was slope one tested with negatives?

Daniel Lemire Slope one is scale and translation invariant, see my article about the concept: http://is.gd/lXc8

Transcript of conversation

Appendix E

SQL Implementation

This Appendix contains the *SQL* implementation used in the tests detailed in Chapter 7.

```
INSERT INTO deviations(pivot_id, deviant_id, value)
SELECT
  ratings.content_id as pivot_id,
  ratings2.content_id as deviant_id,
  AVG(ratings.rate - ratings2.rate) as value
FROM ratings
INNER JOIN ratings as ratings2
  ON ratings.viewer_id = ratings2.viewer_id
  AND ratings.viewer_type = ratings2.viewer_type
  AND ratings.content_id < ratings2.content_id
GROUP BY pivot_id, deviant_id
```

Appendix F

SQL Batches implementation

This Appendix contains the *SQL Batches* implementation used in the tests detailed in Chapter 7. This query is run once per item, and it uses the *id* the current item, indicated here by `{CURRENT_ID}`.

```
INSERT INTO deviations(pivot_id, deviant_id, value)
SELECT
  {CURRENT_ID} as pivot_id,
  ratings2.content_id as deviant_id,
  AVG(ratings.rate - ratings2.rate) as value
FROM ratings
INNER JOIN ratings as ratings2
  ON ratings.viewer_id = ratings2.viewer_id
  AND ratings.viewer_type = ratings2.viewer_type
  AND ratings.content_id = {CURRENT_ID}
  AND ratings.content_id < ratings2.content_id
GROUP BY deviant_id
```

Appendix G

MySQL Configuration

This appendix contains the MySQL configuration used in the tests discussed on Chapter 7.

```
[client]
port = 3306
socket = /var/run/mysqld/mysqld.sock
max_allowed_packet = 32M

[mysqld]
skip-locking
skip-bdb
user          = mysql
pid-file     = /var/run/mysqld/mysqld.pid
socket     = /var/run/mysqld/mysqld.sock
port       = 3306
basedir     = /usr
datadir     = /var/lib/mysql
language     = /usr/share/mysql/english
log-error     = /var/log/mysql.log
bind-address  = 0.0.0.0

# character sets
character_set_server  = utf8
collation_server      = utf8_general_ci

# innodb options
innodb_additional_mem_pool_size = 16M
innodb_buffer_pool_size         = 256M
innodb_data_file_path           = ibdata1:10M:autoextend
innodb_data_home_dir            = /var/lib/mysql
innodb_file_io_threads          = 4
innodb_thread_concurrency       = 4
innodb_flush_log_at_trx_commit  = 2
innodb_log_buffer_size          = 64M
innodb_log_file_size            = 80M
innodb_log_files_in_group       = 3
```

```
innodb_log_group_home_dir       = /var/lib/mysql
innodb_max_dirty_pages_pct      = 90
innodb_lock_wait_timeout        = 120

# myisam
key_buffer_size = 16M

# general
connect_timeout       = 10
back_log              = 50
max_connections       = 96
max_connect_errors    = 10
table_cache           = 2048
max_allowed_packet    = 32M
open_files_limit      = 1024
max_heap_table_size   = 128M
tmp_table_size        = 256M
tmpdir                = /tmp
join_buffer_size      = 4M
read_buffer_size      = 4M
sort_buffer_size      = 8M
read_rnd_buffer_size  = 8M
thread_cache_size     = 8
thread_concurrency    = 8
query_cache_size      = 128M
query_cache_limit     = 2M
thread_stack          = 192K
transaction_isolation = READ-COMMITTED

[mysqldump]
quick
max_allowed_packet = 16M

[mysql]
no-auto-rehash

[myisamchk]
key_buffer        = 64M
sort_buffer_size  = 64M
read_buffer       = 2M
write_buffer      = 2M

[mysqlhotcopy]
interactive-timeout

[mysqld_safe]
open-files-limit = 8192
```