Twilm

TDT4501 - Computer Science, Specialization Project

Datamining Twitter for Movie Recommendation

Authors:
Fredrik Persen Fostvedt
Martin Havig

Supervisor:
Heri Ramampiaro

Norwegian University of Science and Technology,
Department of Computer and Information Science

Acknowledgments

This report serves as the primary piece of documentation of our work during the course TDT4501 Computer Science, Specialization Project, hosted at the Department of Computer and Information Science at the The Norwegian University of Science and Technology, during autumn in the year of 2013.

We would like to thank our supervisor, Heri Ramampiaro, and the customer support at Twitter for their help.

Abstract

This paper explores how the data from Twitter can be used to supplement the Netflix Prize data, and used for movie recommendations. Twitter is a social media network with approximately 9 100 posts per second. Netflix Prize released a dataset with 100 480 507 ratings divided on 17 770 movies. The project explores how these two sets of data can be combined to provided better movie recommendations.

Goals
Understand how data from Twitter can be used for recommendations and predictions.
Implement harvesting of data from Twitter.
Gather data from Twitter.
Supplement the Netflix Prize dataset with data from Twitter.
Implement a Netflix Prize movie rating prediction system.
Compare RMSE scores.

The project evaluates research on the Netflix Prize competition and algorithms used to solve the Netflix Prize dataset. It has been found that the most accurate predictions are not always the preferred ones. Performance plays a big role and is a trade off that should not be made for higher accuracy in a recommendation system. Light running algorithms like k-NN might be preferred, and is subject for future work.

Different designs are proposed in this paper. This includes designs of a framework for harvesting data from Twitter, building a dataset from the gathered data and using the dataset to make predictions on the Netflix Prize dataset.

The APIs provided by Twitter have been found to not be sufficient for creating a dataset that can be combined with the Netflix Prize dataset to make better recommendations. The reason is that the time it takes years to gather such a dataset from Twitter. Alternative directions such as scraping are examined. A scraping approach would be much faster than using the Twitter API. This approach is not allowed by Twitter without their consent.

The project also reflects on the possibilities of future work in the case where a sufficient dataset can be obtained from Twitter. It also reflects on the possibilities of expanding the framework designed for harvesting to a more general framework.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

This chapter will give an overview of what the project is about, and the purpose and motivation behind the project. It will outline the different chapters, and make it clear to the reader where different parts of the project is located in the paper. The paper serves as a documentation of the work done.

1.1 Purpose

Social media data is used frequently in recommendation systems. There are many ways in which this can be done, both in terms of what data is used and what is recommended. The purpose of the project is to explore one of these possibilities, namely how data from Twitter can be used to predict movie ratings and recommend movies. Also, the purpose is to design a system that can use data from Twitter in such a way. Furthermore, the report is meant to ease further development of such a system and explain reasons to why some directions are preferred over other directions.

Goals
Understand how data from Twitter can be used for recommendations and predictions.
Implement harvesting of data from Twitter.
Gather data from Twitter.
Supplement the Netflix Prize dataset with data from Twitter.
Implement a Netflix Prize movie rating prediction system.
Compare RMSE scores.

Table 1.1: The main goals of the research

1.2 Motivation

Social media networks are well established systems. They harbor and produce huge amounts of data on a daily basis. On Twitter, a social media network, 9 100 messages are posted every second [47], and on Facebook, another social media network, 1 million links are shared every 20 minuets [9]. Predictions and recommendations based on this data can be done by gathering some of it and running information retrieval algorithms on it. Several of these social media networks have APIs and ways of making the data they produce accessible to developers. One of the potential ways this data can be used is for movie rating predictions. Hence the motivation for utilizing this huge amount of data for movie rating predictions.

1.3 Context

The following section describes how the project is structured. Table 1.2 gives a brief overview of the content of the different chapters in the report.

Chapter	Description
Chapter 1	The Introduction chapter gives an overview of the project to the reader. It also outlines the purpose and motivation of the project.
Chapter 2	The Preliminary Study chapter documents knowledge, research and technology that is relevant to the project, and how and why some of them were prioritized over others when it comes to how they are used in the project. Twitter and the Netflix Prize dataset is documented in this chapter.
Chapter 3	The Requirements chapter describes the requirements of the project. It also describes how and why they were created.
Chapter 4	The Design chapter describes the design of the system and how it was made.
Chapter 5	The Implementation chapter describes the implementation of the system. Mainly, the implementation of the system harvesting data from Twitter is described.
Chapter 6	Evaluation chapter discussed the development process, testing of results and major issues.
Chapter 7	The Conclusion chapter sums up the project and describes the findings and reflects on them. It also describes further work to be done.
Appendix	The appendix contains extended information such as a full list of the requirements.

Table 1.2: Structure and chapters of the report.

Chapter 2

Preliminary Study

Contents

This chapter documents the various technologies, research and information that are relevant to the project goals.

Beyond this, the study will serve to outline which direction the project should take and what methodologies should be used. Since the span of possibilities and directions is broad and few studies with similar goals and sufficient quality and completeness exist, the study is lengthy and a large part of the project.

2.1 Netflix

This is the prestudy for the movie recommendation and movie rating prediction part of the system. It describes what Netflix is, its datasets, its statistics, and how to use them it.

About Netflix

Netflix is an subscription based on-demand Internet media streaming service. It started out as a DVD-rental business, but changed to an Internet based business in 1999 and has since then been very successful in the media industry. To improve customer satisfaction they developed a personalized movie recommendation system called Cinematch. About 60% of the Neflix users select their next movie or TV-show based on these recommendations [40], so it is important that the system manages to capture the users movie and TV-show preferences and make good predictions about what the users want to watch.

Cinematch

Cinematch is a self-updating recommendation system. It works by searching the Cinematch database for users who have rated the same movie. Then, it finds the non-overlapping movie ratings for these users. At last, it calculates the likelihood that one of the users might like a movie that the user has not seen but that has been rated by the other user.

2.1.1 The Netflix Prize Dataset

Netflix held a competition to help Netflix improve their movie recommendation system. The team that could beat Netflix's own collaborative filtering system by more than 10% won one million dollars. To measure the score root-mean-square error RMSE was used. RMSE is used to measure the difference between the values predicted or estimated and the actual values observed. The RMSE of Cinematch 2.1 was at 0.9525.

Root-mean-square error RMSE

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{n}} \tag{2.1}$$

The equation for calculating the RMSE is show in figure 2.1. It calculates how far the estimated value is from the actual value. Here y_i is the actual value and $\hat{y}$ is the estimated value. The square difference of all the values are averaged by dividing by n. This produces the average distance of all the estimate values from the all the actual values [1].

Competition

For this competition Netflix produced a training dataset based on user ratings from their own database. It contains 100 480 507 ratings by 480 189 unique users of 17 770 movies. The data is split up into 17 770 files, one for each movie, where the data is a triplet of the form <user, rating, date-of-rating>. The user-field and rating-field is an integer, while the date-of-rating-field is on the ISO 8601 form, year-month-day. The number of ratings is different from movie to movie and from user to user.

With the training dataset, the teams system were supposed to predict ratings on a qualifying dataset, which contains 2 817 131 triplets with only user ids, movie ids and dates. This qualifying dataset must be disregarded since the actual ratings for this dataset was only know to the jury of the competition and is nowhere to be found. Instead, a probe set is provided with 1 408 385 users which can be used to remove these from the training set to test predictions based on the probe set. This probe set possesses similar statistical values to the qualifying set. Predictions based on the probe thus gives a good estimate of how the result actually would have scored in the competition [18].

Ratings	100 480 507
Customers	480 189
Movies	17 770
Qualifying set	2 817 131
Probe set	1 408 385

Table 2.1: The data statistics from the dataset provided by Netflix in the Netflix Prize competition

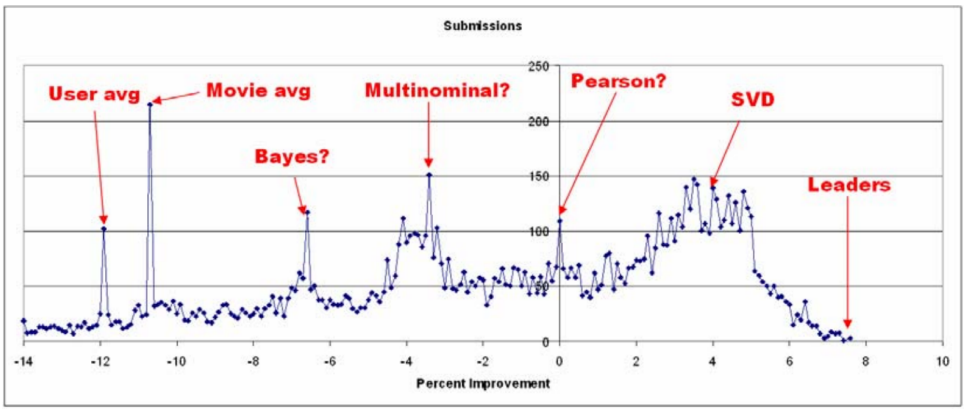

Figure 2.1: Distribution of different submission amounts, their score and the most used algorithms to produce the RMSE score. One of the algorithms with better performance is singular value decomposition (SVD). Average movie rating algorithms were often used, but did not produce good recommendations, reducing the Cinematch score by 10-12%)

Dataset statistics

To better understand the dataset, some statistics about the dataset were produced.

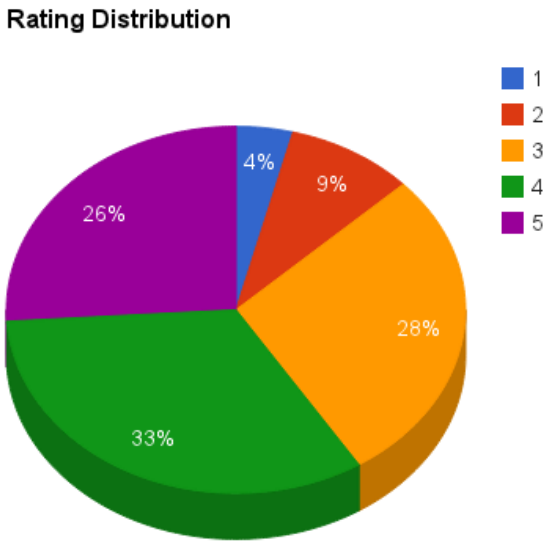

Figure 2.2: Distribution of movie ratings given

The distribution of ratings given is shown in figure 2.2. One third of the ratings given are 4. Ratings 1 and 2 only make up for 13% of the rating distribution. 3 and 5 are quite similarly represented with 28% and 26% respectively. It can from this be assumed that higher ratings should be predicted more often than not.

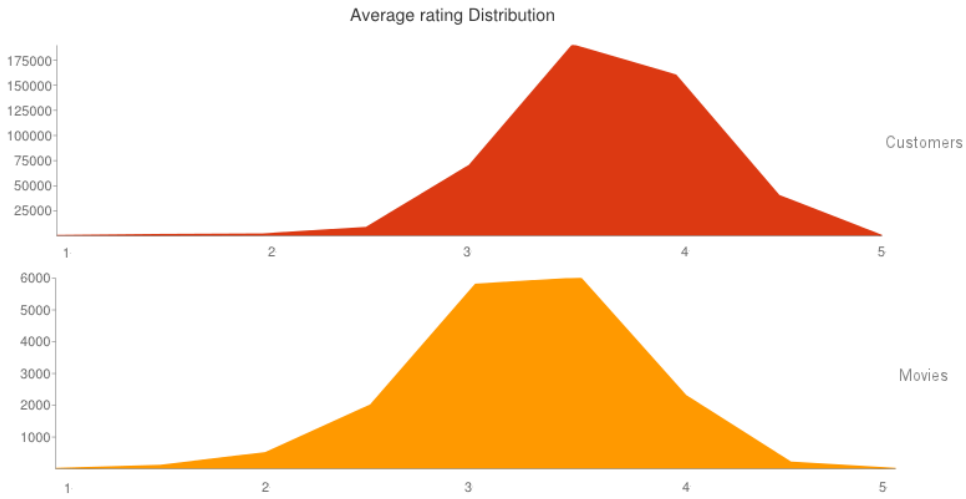

Figure 2.3: Average rating distribution

Average rating distribution is shown in figure 2.3. The upper red graph shows average rating distri-

bution of the users, while the bottom orange graph shows the distribution per movie. Both graphs have quite similar form. Both customers and movies peak at a rating of 3,5, which is natural. The second highest point is for the customers at 4, and for the movies at 3. The actual average user rating of all movies is closer to 3.7.

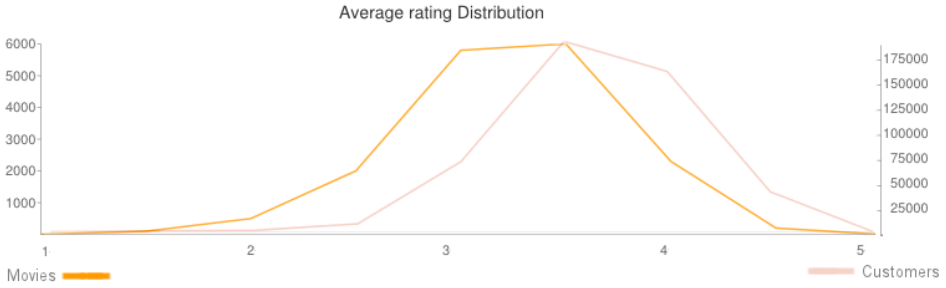

Figure 2.4: Average rating distribution

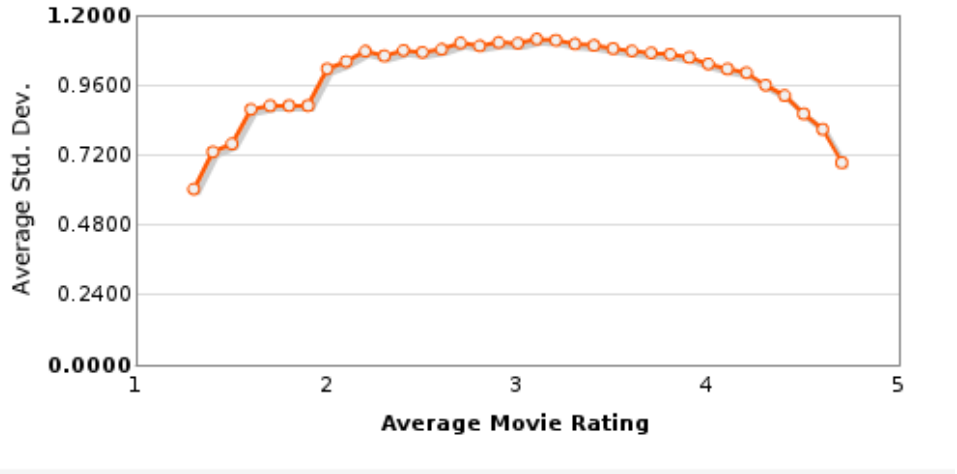

Figure 2.5: Average movie rating distribution [39]

Average movie rating distribution is shown in figure 2.5. For the edge ratings 1 and 5 have the least standard deviation. This implies that when a movie has many ratings, it usually is rated high. The same is implied for low ratings, while for the middle ratings 2 to 4, it is evident that the ratings variate more, and quite similarly for each rating. This implies that the middle ratings are harder to classify. More precisely, when a movie has a rating of 3 the average rating-shift is almost 1.1.

2.2 Twitter

Twitter is a free social media network based on microblogging. In microblogging, a user can post a paragraph of a certain number of characters or less to his/her timeline. The user can also follow other users. Whenever the users that a given user is following posts something, the user can see these posts in his/her feed. This is the basis for many of todays most popular social media networks.

2.2.1 Terminology

This section defines terminology related to Twitter. All words in **bold**
are definitions constructed for this project in particular for more easily describing how the various Twitter terms relate to each other.

Tweet

A tweet is a post containing max 140 characters. It can also refer to the verb 'to tweet', which means 'to post a tweet'. **tweets** is an attribute of a user which returns a list of tweets that the users

user.tweets returns [<tweet>]

User

A user is someone with a Twitter account. A user can post tweets. By following other users, Twitter generates a feed for the user containing the tweets of all the users they follow. They also have certain information retrieval tools available, such as filtering tweets by a tag name, specific user or search term.

Tweeter

A user who tweets a tweet that is referenced in the same context. For example: The best tweets are tweeted by the most popular tweeters

Follower

A follower is someone who follows a user. The follower will see the users tweets in their feed. **followers** is an attribute of a user which returns the list of users that are following that user.

user.followers returns [<user>]

Followee

A followee is someone who a user is following. The user will appear in the followees list of followers, and the followee will appear in the users list with the name following. The following list for a user will be referred to as followees for clarity. **followees** is an attribute of a user which returns the list of users that the user is following.

user.followees returns [<user>]

Hashtag

Hashtags are topic labels that are placed on tweets by the tweeter. They are made by typing #tagname anywhere in the tweet when composing it. By navigating to #tagname, a user will see a feed of all tweets containing #tagname. **hashtags** is a an attribute of a tweet which returns the list of hashtags that the tweet contains.

tweet.hashtags returns [<hashtag>]

Hashtag also has the attribute tweets, which returns all tweets that have a certain hashtag. These hashtags are ordered by Twitter's algorithm which balances how recent the tweet is with the popularity of the tweeter.

hashtag.tweets returns [<tweet>]

Usertag

Usertags are labels that are placed on tweets describing who the tweets are addressed to. They are made by typing @username anywhere in the tweet. This tweet will also show up in the feed of the user with name username. **usertags** is an attribute of a tweet which returns the list of users that the tweet is addressed to.

tweet.usertags returns [<user>]

2.2.2 Legal Considerations

Collecting and using a data set from Twitter, a company whose currency is information, must be done in a way that complies with their rules and terms of service. This section considers the legal considerations of gathering data from Twitter with an automated system.

API Terms

You may use the Twitter API and Twitter Content in connection with the products or services you provide (your "Service") to search, display, analyze, retrieve, view, and submit information to or on Twitter [42]. This means that data can be gathered from Twitter and analyzed as long as the Twitter API is being used.

If you provide downloadable datasets of Twitter Content or an API that returns Twitter Content, you may only return IDs (including tweet IDs and user IDs) [42]. A system that utilizes the API to gather and analyze a dataset can be built and run legally as long as the dataset is not made available, or the dataset only contains IDs.

General Terms of Service

Crawling the Services is permissible if done in accordance with the provisions of the file twitter.com/robots.txt [45]. However, scraping the Services without the prior consent of Twitter is expressively prohibited [46]. It is thus possible to crawl Twitter as long as the page is only being indexed and it is done in accordance with twitter.com/robots.txt. Scraping the pages for tweets and other data can only be done with the prior consent of Twitter. Twitter has been contacted about whether or not data can be scraped for analysis in context of movie recommendations for this project. A decision maker with the authority to grant this kind of consent at Twitter has not replied to the request.

2.2.3 REST API

Twitter's HTTP/HTTPS based REST API allows the application using it to perform many of the core functionalities of Twitter. This API fully reflects Twitter's content.

Usage

The API is divided into POST and GET requests. The POST requests makes changes to Twitter. A POST request could be made for posting a tweet. Another POST request could be made to update a user's profile. The GET requests retrieves data from Twitter. A GET request could be made to retrieve the followees or followers of a user. Another GET request could be made retrieve the tweets on the time line of a user. The data is returned as JSON. [44]

Authentication

Twitter's REST API uses an access token generated for the specific application in order to make a request. In addition, OAuth can be used if the session is on behalf of a specific user. Some requests such as posting a tweet can of course only be made on behalf of a user and thus requires OAuth. [44]

Rate Limit

The number of requests an application or a user can make is limited to a 15 minute window. Within this 15 minute window the each user can make 15 GET requests. This means that a user can retrieve the followees of 15 users withing the 15 minute window, but must wait until the 15 minutes have passed before retrieving more data. The effective request rate limit is 1 request per minute. This means that large amounts of data can only be retrieved over long periods of time.

If only the application access token is used and not OAuth, the application as a whole can make 15 GET requests within the 15 minute window. [43]

2.2.4 Search API

Twitter's Search API is a REST API that is used for searching for tweets that match a search query.

Usage

The Search API uses only GET requests. Each GET request contains the search query as well as other parameters such as location to specify the search further. The API will return tweets that contain all the words in the search query in any order. The tweets are returned as JSON. [44]

It is important to note that not all tweets are indexed in the Search API. Only a selected number of tweets within the time frame of one week are available. The tweets returned by the Search API are thus only a subset of all tweets. [41]

Authentication

To authenticate, the access token generated for the application can be used as well as OAuth. This is the same as the REST API. [44]

Rate Limit

The number of requests that can be made are limited to 180 requests per 15 minute window for each OAuth. In effect, the request rate limit is 12 requests per minute. If the Search API is accessed using only the application access token, these limitations apply application-wide for any number of users. [43]

2.2.5 Stream API

Twitter's Stream API streams data from Twitter as it is created to a client through a persistent HTTPS connection. It is ideal for gathering the most up-to-date tweets or gathering large amounts of data over long periods of time.

Usage

The client makes an initial GET request specifying the type of data it wants to receive. The request contains a list of up to 400 keywords. Any tweet that matches one or more of these keywords are considered part of the stream. The request can be directed to two endpoints: A public endpoints, which streams tweets from all users; or a user endpoint, which streams tweets from a specific user.

After the initial request is made, a persistent HTTP connection is opened. Then, the tweets that match the keywords in the initial request are streamed. The tweets keep streaming until the connection closes.

Authentication

Authentication for using the Stream API is done with OAuth. There is no application wide identification such as in the REST API and the Search API.

Limitations

There is no rate limit in the Stream API. However, there are a few cases in which the streaming will stop. For instance, If the client fails to receive the streamed data and Twitter's buffer builds up too many tweets, the connection will be closed. Also, if the client opens too many streams with the same credentials, the oldest connection will be closed.

It also takes the Stream API a very long time to create a dense dataset. By a few searches for movie titles that are less watched, it can easily be determined that many of them have only a few tweets posted about them over the span of years. This means that it would take at least a year for the Stream API to deliver any data points about the movie.

Testing API Capability

To further understand what kind of dataset the Stream API would produce over time, a sample of 20 random movies were selected from the Netflix Prize dataset. The mongoDB console command used to retrieve this sample can be seen in listing 2.1. Then, the movie titles and release year were searched for using Twitter web search. Values were recorded for how many tweets were available for a search for a given movie within the last month, half a year and a full year. Values for how many days ago the last tweet for each movie was posted were also recorded. The results can be seen in table 2.2

```
db.netflix_movies.find().limit(-1).skip(Math.random()*18000).next()
```

Listing 2.1: The mongoDB console command used to generate a random selection of movies

The data recorded in table 2.2 was then aggregated to show the coverage of tweets for a given time period. The aggregation can be seen in table 2.2. If the Stream API is run for one year, about 65% of

the movies in the Netflix dataset will have one tweet or more. If more than 8 tweets are needed for a movie, the Stream API will only produce this number of tweets for about 45% of the movies.

Movie Title	**Year**	**Days since last tweet**	**Tweets last month**	**Tweets last half year**	**Tweets last year**
Mickey Blue Eyes	1999	5	1	16	26
The Marquise of O	1976	162	0	1	2
The James Bond Story	1999	176	0	1	5
Gift of Love	1999	171	0	3	3
Miami Vice: Season 1	1984	7	1	1	3
Silent Running	1971	9	1	12	31
Bjork: All Is Full of Love	1999	2	6	103	114
Koyaanisqatsi: Life Out of Balance	1983	3	1	12	13
Variable Geo	2003	801	0	0	0
The Man from Laramie	1955	7	2	15	28
The Twilight Zone: Vol. 21	1963	401	0	0	0
In the City	2003	700	0	0	0
Midsomer Murders: Beyond the Grave	2000	715	0	0	0
A Nightmare on Elm Street 5: The Dream Child	1989	6	4	27	48
Time Changer	2002	49	0	2	11
That Old Feeling	1997	129	0	7	12
A Loving Father	2002	274	0	0	1
Absolutely Fabulous: Absolutely Special	1996		0	0	0
Ghost Story	1981	2	7	31	44
Hera Pheri	1975		0	0	0

Table 2.2: Data gathered from Twitter web search about a random selection of 20 movies from the Netflix Prize Dataset.

% of movies where tweets found was..	**Last month**	**Last half year**	**Last year**
> 1	20.00%	50.00%	65.00%
> 2	15.00%	45.00%	60.00%
> 4	10.00%	40.00%	50.00%
> 6	5.00%	40.00%	45.00%
> 8	0.00%	35.00%	45.00%
> 16	0.00%	15.00%	30.00%
> 32	0.00%	5.00%	15.00%
> 64	0.00%	5.00%	5.00%
> 100	0.00%	5.00%	5.00%

Table 2.3: Aggregation of the data in table 2.2 showing the percentage of how many movies had more tweets than a given number for a given time period.

2.2.6 Crawling and Scraping Twitter

This section will examine how to crawl and scrape Twitter's Web Search. This will be done by browsing twitter.com and examining the pages in Google Chrome Developer Tools GCDT [14] as the site is being browsed and search requests are being made. The goal of this examination is to discover requests a web crawler must be capable of making in order to search Twitter and harvest tweets and users automatically and how a scraper can distinguish the DOM elements that describe a tweet.

Search Request

Tweets can be found using an HTTP search request with the search query in the format URLE [34]. Searching for <URLE search query> can be done with the HTTP request is shown in listing 2.2. In the rest of the discussion, the search query "The Dark Knight Rises 2012" will be used. The search request for this search query parsed to URLE is shown in listing 2.3

```
http://twitter.com/search?q=<URLE-search-query>&src=typd&f=realtime
```

Listing 2.2: URL of a Twitter HTTP search request for <URLE-search-query>

```
http://twitter.com/search?q=the%20dark%20knight%202012&src=typd&f=realtime
```

Listing 2.3: URL of a Twitter HTTP search request for "The Dark Knight Rises 2012" parsed to URLE

Using the GCDT [14] the DOM element on the search page that contains all the tweets that have been returned can be found using the select element tool. The HTML of this element is shown in listing 2.4. Within it, each tweet is contained in a DOM element like the one in listing 2.5

```
<ol class="stream-items js-navigable-stream" id="stream-items-id\">
```

Listing 2.4: HTML of the DOM element containing all tweet elements

```
<li class="js-stream-item stream-item stream-item expanding-stream-item" data-item-id
    ="397530125715402752" id="stream-item-tweet-397530125715402752" data-item-type="
    tweet">...</li>
```

Listing 2.5: HTML of a tweet element

Using jQuery [20] and the JS Console in GCDT [14], it is possible to determine how many tweets have been returned. This is done by defining JS function that selects the tweets elements and returns the number of elements in this list in listing 2.6. The function can be assured to work by counting the number of tweets displayed on the search result page and making sure that it matches the number returned by the function.

```
> number_of_tweets_in_search_results = function(){ return $("li[data-item-type='tweet
    ']").size(); }
> number_of_tweets_in_search_results();
15
```

Listing 2.6: Creating a function in GCDT JS Console for counting the occurance of tweets on the Twitter search result page

Scroll Search Results

In order to obtain more search results, it is necessary to scroll to the bottom of the page for the browser to make an XHR request to retrieve more tweets. As the scrolling is done, XHR requests can be monitored in the Network Console of GCDT. Consecutive XHR requests are sent as scrolling is done several times. This is shown in listing 2.7

```
JS Console:
> number_of_tweets_in_search_results();
15

(Scrolling to the bottom of the page)

Network Console > XHR:
Request URL:https://twitter.com/i/search/timeline?q=the%20dark%20knight%202012&src=
    typd&include_available_features=1&include_entities=1&last_note_ts=0&scroll_cursor=
    TWEET-360758159364734978-399762568459612160


JS Console:
> number_of_tweets_in_search_results();
35

(Scrolling to the bottom of the page)

Network Console > XHR:
Request URL:https://twitter.com/i/search/timeline?q=the%20dark%20knight%202012&src=
    typd&include_available_features=1&include_entities=1&last_note_ts=0&
    oldest_unread_id=0&scroll_cursor=TWEET-338938038694584320-399762568459612160

JS Console:
> number_of_tweets_in_search_results();
50

(Scrolling to the bottom of the page)

Network Console > XHR:
Request URL:https://twitter.com/i/search/timeline?q=the%20dark%20knight%202012&src=
    typd&include_available_features=1&include_entities=1&last_note_ts=0&
    oldest_unread_id=0&scroll_cursor=TWEET-316893896573607937-399762568459612160

JS Console:
> number_of_tweets_in_search_results();
65
```

Listing 2.7: GCDT JS Console and Network Console showing the effects of scrolling to the bottom of the Twitter search result page several times.

As the requests are sent and processed, tweets are appended to the tweets element. All that is needed in order to keep receiving tweets is to keep sending requests of this pattern. The attributes of the XHR requests consist of the variables and constants shown in listing 2.8

```
constant src=typd
constant include_available_features=1
constant include_entities=1
constant last_note_ts=0
constant oldest_unread_id=0
variable scroll_cursor=TWEET-<18 character number>-<18 character number>
```

Listing 2.8: Variables and constants of the XHR requests sent when scrolling to the bottom of the Twitter search result page.

The only attribute that seems to variate with each request is scroll_cursor attribute. This means that the piece of JS code that is making the requests somehow has access to the information contained in this changing attribute. A DOM tree search for 'TWEET-' shows that scroll_cursor is contained by a div with the class stream-container under the attribute 'data-scroll-cursor' and 'data-refresh-cursor' as seen in listing 2.9

```
<div class="stream-container" data-scroll-cursor="TWEET
    -311337362556846081-399762568459612160" data-refresh-cursor="TWEET
    -396773826303762432-399762568459612160">
```

Listing 2.9: The div element of the Twitter search result that contains the scroll cursor which is used in the XHR requests made to retrieve more tweets when scrolling to the bottom of the page.

By scrolling to the bottom of the page again, a request with scroll-cursor information from one of these attributes is sent. As shown in listing 2.10, the attribute data-scroll-cursor is responsible for what goes in the attribute scroll-cursor.

```
Network Console > XHR:
https://twitter.com/i/search/timeline?q=the%20dark%20knight%202012&src=typd&
    include_available_features=1&include_entities=1&last_note_ts=0&scroll_cursor=TWEET
    -311337362556846081-399762568459612160
```

Listing 2.10: XHR request made when scrolling to the bottom of the page showing that the scroll cursor has been extracted from the div element in listing 2.9

Scraping Search Results

Each tweet element in the search results contains DOM elements of importance that can be scraped. These DOM elements are shown in table 2.4.

Element Name	Content
User Name	The name of the user who posted the tweet
User ID	The ID of the user who posted the tweet
Tweet Text	The text contained in the tweet

Table 2.4: Elements to be scraped from each tweet element that has been retrieved

User Name

The user name can be found in the header of the tweet. Using GCDT Select, it can be extracted from the page as shown in blue in figure 2.6. It will be referred to as the user name element. The HTML of this element is shown in listing 2.11.

Figure 2.6: User name element (shown in blue) in a tweet retrieved by Twitter web search

```
<span class=\"username js-action-profile-name\"><s>@</s><b>BatmanMemorabil</b></span>
```

Listing 2.11: HTML of the user name element in a tweet

Using jQuery, a function can be created to count the occurrences of the class "username js-action-profile-name", as shown in listing 2.12. This function can then be used to compare the number of user name elements in the DOM tree to the number of tweets in the search result, as shown in listing 2.13. Since they are equal, it is highly likely that the right class is being scraped for the user name element.

```
> number_of_tweet_user_name_element_occurances = function(){ return $("span.username.
    js-action-profile-name").size() }
```

Listing 2.12: Creating a function in GCDT JS Console for counting the occurance of user name elements on the Twitter search result page

```
> number_of_tweet_user_name_element_occurances();
119
> number_of_tweets_in_search_results();
119
```

Listing 2.13: Running functions in GCDT JS Console to show that the number of user name elements matches the number of tweets

User ID

The user ID can also be found in the header. It is contained in the attribute data-user-id. Using GCDT Select, it can be scraped from the page. It will be referred to as the user id element. The HTML of this scraping is shown in listing 2.14

```
<a class="account-group js-account-group js-action-profile js-user-profile-link js-nav
    " href="/username" data-user-id="userid">...</a>
```

Listing 2.14: HTML of the user id element in a tweet

Using jQuery, a function can be created to count the occurrences of the class used to specify the user id element, as shown in listing 2.15. This function can then be used to compare the number of user id elements in the DOM tree to the number of tweets in the search result, as shown in listing 2.16. Since they are equal, it is highly likely that the right class is being scraped for the user id element.

```
> number_of_tweet_user_id_element_occurances = function(){ return $("a.account-group.
    js-account-group.js-action-profile.js-user-profile-link.js-nav").size() }
```

Listing 2.15: Creating a function in GCDT JS Console for counting the occurance of user id elements on the Twitter search result page

```
> number_of_tweet_user_id_element_occurances();
21
> number_of_tweets_in_search_results();
21
```

Listing 2.16: Running functions in GCDT JS Console to show that the number of user id elements matches the number of tweets

Tweet Text

The tweet text can be found in the content of the tweet, as shown in blue in figure 2.7. Using GCDT Select, it can be extracted from the page. It will be referred to as the tweet content element. The tweet content element contains several HTML elements from which the text can be scraped, as shown in listing 2.17.

Figure 2.7: Tweet content element (shown in blue) in a tweet retrieved by Twitter web search

```
<p class="js-tweet-text tweet-text">
  <strong>The Dark Knight</strong> Rises (DVD, <strong>2012</strong>)
  <a href="http://t.co/p8pr9wqC4k" rel="nofollow" dir="ltr" data-expanded-url="http://
      ift.tt/17i5F5V" class="twitter-timeline-link" target="_blank" title="http://ift.
      tt/17i5F5V">
    <span class="tco-ellipsis"></span>
    <span class="invisible">http://</span>
    <span class="js-display-url">ift.tt/17i5F5V</span>
    <span class="invisible"></span>
    <span class="tco-ellipsis">
      <span class="invisible"> </span>
    </span>
  </a>
  <a href="/search?q=%23batman&amp;src=hash" data-query-source="hashtag_click" class="
      twitter-hashtag pretty-link js-nav" dir="ltr">
    <s>#</s><b>batman</b>
  </a>
</p>
```

Listing 2.17: HTML of the tweet content element in a tweet

Using jQuery, a function can be created to count the occurrences of the class "js-tweet-text tweet-text", which most likely describes all tweet content elements. The function is shown in listing 2.18. By comparing this number with the number of tweets in the search results and making sure it is equal, it can be verified that it is highly likely that class is describing the correct DOM element. The comparison is shown in listing 2.19

```
> number_of_tweet_content_element_occurances = function(){ return $("p.js-tweet-text.
    tweet-text").size() }
```

Listing 2.18: Creating a function in GCDT JS Console for counting the occurance of tweet content elements on the Twitter search result page

```
> number_of_tweet_content_element_occurances();
119
> number_of_tweets_in_search_results();
119
```

Listing 2.19: Running functions in GCDT JS Console to show that the number of tweet content elements matches the number of tweets

First, in order to obtain the tweet text, everything that is not a link or contained in a link must be scraped from the tweet content element in listing 2.17. An example of the HTML of this scraping is shown in listing 2.20. Then, the tweet text is obtained by stripping the HTML of all tags.

```
<strong>The Dark Knight</strong> Rises (DVD, <strong>2012</strong>)
```

Listing 2.20: The HTML of scraping everything from listing 2.17 that is not a link

2.3 Similar Systems

In this section, similar systems will be discussed. Their usability in the project will also be evaluated.

2.3.1 Recommender Systems

MovieLens

MovieLens is a movie recommendation system. A user creates an account and gets movie recommendations based on the user's own ratings through MovieLens's collaborative filtering algorithm. The algorithm uses an item-item based algorithm. It takes the 20 movies which are most similar to the target movie and looks at how these 20 movies were rated. [16]

The issue with using a strict item-item based correlation and the MovieLens recommendation system is that for a user with a low rating average, the user will very seldom get recommended movies with a top to average rating.

MovieTweetings

MovieTweetings is an application for fetching movie ratings through the Twitter API from Twitter. The application searches for ratings shared from IMDB and ratings that are already stored by the application. On IMDB, users have the option of sharing ratings on Twitter. If they do, the rating will be caught by the MovieTweetings application.

Their motivation for using a social media dataset rather than using Netflix Prize dataset or MovieLens's datasets 2.3.1 is that they wanted a dataset that was more up to date. At the time of writing the article, they had more than 60 000 ratings, and about 500 new were added every day [35]. The Twitter API is queried daily to update their database.

While querying the API for well structured tweets with ratings about movies is a good approach when searching for a database with movie ratings, the data will be sparse. The data is not based on the users followers and followees, and will therefore not represent the social strength Twitter is capable of producing. A direct access to IMDB, rather than going through Twitter to get the rating information also seems like a simpler and more result-producing way to accomplish the same task.

Twittomender

Twittomender is a followee recommender for Twitter. Twitter uses a collaborative filtering approach for recommending new followee. This does not necessarily produce the most relevant recommendations for a user. Twittomender takes a different but similar approach to the problem. Tweets of a user are used and compare with other users tweets to to find similarities and recommend new Twitter-friendships [19]. The 7 recommendation strategies used by Twittomender are shown in table 2.5.

Type	Strategy
Content-based	(1) Users own tweets (2) Followee's tweets (3) Follower's tweets (4) All tweets
Collaborative-based	(5) Followee's Ids (6) Follower's Ids (7) All Ids

Table 2.5: The 7 recommendation strategies used by Twittomender

The user interacts with the system through providing the system with their username on Twitter. The system then produces a profile of the user based on tweets, and determines a score against other users based on most frequent terms amongst the users tweets. Each of the strategies from table 2.5 are used and top 20 from each are merged to a list where the most frequent users are returned as recommended followees.

After testing on 1000 users the best performing strategy was a collaborative-based approach, using a users follower connections for recommendation

What can be concluded from this is that there is data in a user's relationships that indicates interest or more precisely a willingness to subscribe. In particular for followers recommended as followees.

2.3.2 What the top Netflix Prize contestants did

BellKor's Pragmatic Chaos

BellKor's Pragmatic Chaos, the Prize winners of the Netflix Prize competition, was a group of scientists from AT&T Labs, Yahoo!, Pragmatic Theory and Commendo Research & Consulting. They achieved a RMSE of 0.8567, which was 10.06% better than Netflix's own Cinematch score.

Models for the rating the data is learned by fitting previously observed ratings, but since the model is to be used to predict future ratings, overfitting the data must be avoided. This achieved by adding an error to the learned parameter, also known as regularizing the learned parameter. It is achieved by using a regularization constant. This and other constant such as: step size and number of iterations are set manually. The algorithm is run multiple times with different values for the constants, the values used which produces the best score is used. [23]

The winning algorithm was put together based on multiple different modeling techniques. Some of these modeling techniques are described below.

Time of week rating The team saw that ratings made weekends had a tendency to have a higher value than ratings made on for Monday, for instance. The same applies for different times of day. Higher ratings do not necessarily imply that the movie is much better than a movie rated lower another day or at another time of day. The users just happen to rate movies differently at different points in time.

Close User ratings Multiple ratings made in a small interval of time by the same user implied that the user was rating based on the his/her recollection of the rated movies, and not that the user just saw that movie. These ratings had to be handled with respect to one another. This is because of the effect the faulty or old memories had on these ratings. The all time favorites/negatives are also usually in this bulk. While the movies which did not make much of an impression are forgotten. This helps explain

figure 2.5, where movies with the edge ratings of 1 or 5 had a lower rating distribution.

Baseline predictors Some users have the tendency to rate on an overall average higher than other users.

Movie like-ability Movies can go in and out of popularity. This change usually happens over a extended period.

Frequency term The number of ratings a users gives on a specific day can help explain data-variability during that day.

Matrix factorization with Temporal Dynamics (SVD) Single value decomposition use two sets of static movie factors. Set one is used in all the models of the factors. Set two uses the movies rated by the user, normalizes the sum, and uses this as a user factor.

Neighborhood model with Temporal Dynamics The most common approach to collaborative filtering. They used an item-item model based on global optimization. To predict the rating of a new item based on rating an old item two sets of weights were used. One where the values of the ratings were accounted for, and one disregarding rating values. A situation where a user rates an old item and a new item close to each other both in time and rating, is a better indicator that these items can be related to each other than if two items are rated similarly with a greater timespan between each other.Therefore a decaying relationship is introduced between items. This decreased RMSE from 0.9002 to 0.8870. [23]

Clustering In a cluster, the average value is the predictor for the user movie pairs in the cluster.

To construct the clusters the set of movies are randomly divided into 128 bins and the set of users are divided into 128 bins. This creates $128 * 128 = 16384$ clusters. The intra-cluster rating variance was minimized trough moving users and movies between the bins, while keeping the bin amount. This approach was done greedy till there was no significant improvement of the intra-cluster variance. This was their only use of clustering. They found that it had no measurable impact on the accuracy of the final blend [27].

Anchoring Certain collections of movies are watched in a given order. For instance, sequels are usually watched in the order they where made, and should be predicted accordingly.

Interestingly, the bias model that considers the factor that users change rating-scale over time, resulted in a RMSE of 0.9555, which is almost as good as the Cinematch Netflix was already using. The learning with the different modeling techniques is done by a stochastic gradient descent algorithm running for 30 iterations. This is the only time they use an automatic parameter tuner. [33]

The "BellKor's Pragmatic Chaos"-team used a lot of different models and did a lot of tweaking to get to the top. Some interesting findings to take from this is the importance of diversity and how much information about the rating the time stamp contains.

The Ensemble

The Ensemble is a merge of the teams "Grand Prize Team" and "Opera Solutions and Vandelay United". They also managed to get an RMSE of 0.8567, and therefore a 10.09% improvement as well. But since the "BellKor's Pragmatic Chaos"-team managed to submit their results 20 minutes earlier than "The Ensemble"-team, they lost the competition.

The two merged teams that composed "The Ensemble" consisted of multiple smaller teams. They had all competed in the Netflix Prize challenge, and all put their heads together and collaborated to get the kind of results they got

"The Ensemble"-team used a large selection of different models, just as the "BellKor's Pragmatic Chaos"-team.

After the competition was done "The Ensemble" tested a merger of their system with the "BellKor's Pragmatic Chaos"-system. The test RMSE of a 50/50 blend was 0.855476, which is a 10.19% improvement. This is 0.10% better then the teams managed to produce by themselves.

Conclusion

Looking at the two top contestants from the competition forms some interesting pattern. Diversity is key to be able to predict user ratings well.

While both the teams managed to produce a 10% improvement over the original Netflix algorithm, it was never taken into use [49]. Even though SVD and and RBM lowers the RMSE to 0.88 alone, are they not as applicable as wanted. The actual rating amount is 50 times bigger than the training set, and for SVD to give an accurate value, it has to be recalculated for each new rating, which is not applicable in a world where live recommendation is expected [49].

2.3.3 Recommender Algorithms

Probabilistic Neighborhood Selection

Neighborhood-based collaborative filtering is a common approach for recommending items. This approach often has the issue that it over-specializes and makes concentrated biases when recommending. When trying to estimate an unknown value, the nearest neighbors tend to produce a recommendation list with known items. The taste of the user is often multidimensional when it comes to movies, and these can be unknown to the k nearest neighbors to the user [8]. This can be handled with a probabilistic approach when selecting neighbors [30]. The approach does not estimate an unknown value based on the weighted averages of the k nearest neighbors, but rather on k probabilistically selected neighbors [37]. Another way of handling this issue is suggesting the least similar neighbors and their dislikes [3]. This approach was proved to be less accurate when calculating the RMSE [30], and will therefore not be examined further.

KNN is linear in the size of the training set, and therefore an efficient classification algorithm [5]. And different implementations of it managed to produce RMSE scores from 0.91 to 1.002 [21, 22, 38, 50], even though K-NN has show to have some issues with sparse data [15], which implies that it will do better with a more complete dataset, which will be produced by the addition of social media data.

Matrix Factorization

There are several matrix factorization algorithms. Amongst them single value decomposition (SVD) and alternating least squares (ALS) are valid candidates. Bellkor's solution 2.3.2 used single value decomposition. This was a good addition to their solution. The issue with this algorithm is that it is rather computationally expensive and re-computation is needed to be done when new ratings are added, and therefore not very applicable to a live recommendation system. SVD performs well on sparse matrices, which the Netflix Prize dataset is [15].

ALS is a simpler and a parallelizable matrix factorization algorithm with many of the benefits that matrix factorization brings [26].

This algorithm can be used to populate sparse matrices, just like the one from Netflix where the matrix is made up from users and movies. A rating matrix, R, is produced based on the use of two much

smaller matrices. A number of features are set for each of the two smaller matrices U and M, one for the users and one for the movies respectively. The idea is to get $U^\top M$ as close to the real R. U and M are computed in two steps. This is easily parallelizable since solving for U is independent of the other users, therefore the columns of U can be computed in parallel. The information is gathered, and M is updated. There are no computational dependencies between the rows in the movie matrix [29]. This algorithm produced a score of 0.9278, which is a 2% improvement over Cinematch [26]. The runtime to produce this score was 2 hours on a high-end system [17].

2.3.4 Conclusion

There are two basic problems when analyzing very large datasets: Writing the code correctly and the performance (i.e. speed at which it runs). As was shown in 2.3.3, matrix factorization is one of the algorithms which preformed the best on the Netflix Prize dataset. This is also shown in figure 2.1.The issue with it is that if new ratings are to be included in the recommendation system, all recommendations must be recomputed, which is not ideal for a live recommendation system.

The ALS 2.3.3 proved to be a very parallelizable algorithm, and a potential good alternative to SVD when doing live recommendation. However, the algorithm still requires a lot of computational power and will not be feasible to run when wanting live results on a low-end. But could prove to be a potential algorithm to use if one where to scale up to a high-end system.

The probabilistic neighborhood selection is a form of K-NN has a good runtime and produced a respectable RMSE. This is a well know approach to collaborative filtering, and there is a lot of documentations on different ways of selecting neighbors and how to implement the algorithm. This is a good algorithm for producing recommendations on big datasets.

2.4 Development language and technologies

2.4.1 Database

In order to easily retrieve data from the Netflix Prize dataset and store and retrieve data from Twitter, it is useful to consider various kinds persistent storage or databases. There are three categories of options: Traditional file systems, SQL and NoSQL. Since traditional file systems are not designed for dynamically storing and retrieving data based on the structure of the data, they are not considered. The two options remaining are SQL and NoSQL which will be discussed and evaluated in the following sections.

NoSQL

NoSQL databases can be defined as being non-relational. In many cases they are also: Open-source, distributed and horizontally scalable [28]. NoSQL is meant for storing large amounts of data [48], where all the data does not share the same attributes or schema. They are focusing on eventually consistency BASE instead of the more traditional consistent atomic transactions ACID [31]. There are a number of different kinds of NoSQL database classes. The common classes are: key-value, column, document, graph and relational.

Data Model	Performance	Scalability	Flexibility	Complexity	Functionality
Key-value	high	high	high	none	variable (none)
Column	high	high	moderate	low	minimal
Document	high	variable (high)	high	low	variable (low)
Graph	variable	variable	high	high	graph theory
Relational	variable	variable	low	moderate	relational algebra

Table 2.6: Differences in features of NoSQL Database Types

As seen in table 2.6 performance, scalability and flexibility are rated high in all categories. [25, 36]

SQL

SQL databases are the most common way of storing structured data. The data is stored in tables with columns and rows and is schema defined. This means that the attributes of the different data types are already defined and must be set whenever a data point is added to the database, even if it is NULL. The database can make relations between the tables ids if necessary. Hence the name relational databases. Performing queries is the way users and applications modify and read a SQL database. They are simple operations such as SELECT, UPDATE and DELETE written in the SQL query language. On a higher level of abstraction are transactions. Transactions are more or less a collection of queries. SQL Databases follow the ACID(atomicity, consistency, isolation, durability) principles for transactions. This means that the state of the database will never appear partially modified by a transaction to a new transaction that is being executed. Consistency of transactions are great for ensuring consistency across the data, but comes at a performance cost. [32]

Database Alternatives

Recommender systems are in nature performance heavy. They are also statistical which makes them fault tolerant. For this reason performance is valued over consistency. The dataset is likely to have nested document structure as it will combine two different types of data. Being able to append and combine data based on the structure of the content thus has value. For these reasons the alternatives centered around NoSQL document stores, which are fast and have supports operations based on the structure of the document. MySQL, which is the faster of the SQL alternatives, was also considered.

MongoDB

MongoDB is a fast NoSQL document store with high availability. Instead of storing the data in tables as in SQL, the data is stored in a BSON document. BSON stands for Binary JSON and is essentially a JSON document that is stored in binary instead of ASCII. A JSON document is a nested hash which can hold any number of attributes without depending on the other documents in the store. It is thus schemaless. Even though it is schemaless, MongoDB still possesses the some of the great properties from SQL databases, such as indexes, dynamic queries and updates.

MongoDB is implemented in C. It also has sharding which allows the database to scale horizontally. Custom queries and map-reduce queries can be written in JS, which has fast interpreters. This makes mongoDB an efficient and high speed data base for storing, modifying and fetching data with a nested document structure [2]. In addition it has bindings to many different languages, which is a great advan-

tage if some of the modules that need to access the database needs to be written in a language other than what it originally was written in.

CouchDB

CouchDB is similar to MongoDB in the sense that it is a NoSQL document store. It also has the same query capabilities as mongo and also uses JS as a query language. In addition, CouchDB has a HTTP REST API which allows any web application to access it easily. CouchDB has a lot of built in features which makes web development with CouchDB easy. One of the greatest advantages is the intuitive web-based admin user-interface that is provided.

CouchDB does not support sharding which makes it less horizontally scalable. An alternative to achieve this is to use CouchBase, which is the distributed version of CouchDB. CouchDB is implemented in erlang. While erlang is faster than it used to be, it is certainly slower than a similar system written in pure C. In addition, CouchDB stores a copy of every single document every time it is modified. This makes CouchDB grow in size with time. [12, 13]

MySQL

MySQL is one of the most popular databases in the world. This is because of its high performance, reliability and ease of use. It should therefore be considered for the question of which database system to use. MySQL is an SQL based relational database, as opposed to the two NoSQL databases. It has bindings to most languages. It provides a much more advanced high level query language than a typical NoSQL document store. This is great for easily making complex queries during development, which is somewhat more tricky in NoSQL and JS. However it is slower because of the interpretation of this high level language. The SQL database schema makes it more difficult to change the format of the data. MySQL also does not support JSON which is one of the most common standards for data retrieved from the web. This creates extra work with parsing JSON to the schema or invoking a mapper that has been created for this purpose. [**?**]

Conclusion

Even though MySQL has several of the properties needed for a recommendation system, it lacks the performance and flexibility of a NoSQL store, which is a huge advantage. The complex join capabilities are not strictly necessary and can be replicated in JS in any of the two NoSQL document stores.

CouchDB has some unique merits. The simple web UI and HTTP REST API are great for development and compatibility with web applications. However, compared to mongoDB it scores low on performance both as a standalone database, and in the sense that it can not be sharded to increase performance horizontally if necessary. It also takes up more space than necessary and grows in size for every single modification, which is an unfortunate property for a database that should handle a large fault tolerant dataset.

For these reasons mongoDB was chosen as the most appropriate alternative.

2.4.2 Programming languages

The goals of the project involve harvesting large amounts of data from the web, store it quickly in a database and do large computations on this data quickly. Because the design and implementation is likely to go through many iterations, it is important that the language chosen is most of all easy to modify. Performance of the language is also a valuable property.

Ruby

Ruby is a dynamic, interpreted, open source programming language with a focus on simplicity and productivity. Even though it is sometimes difficult to interpret ruby because there are always a large number of different ways to achieve the same functionality, ruby has an elegant syntax that is natural to read and easy to write. This is due to the fact that everything in Ruby is an object. Essential parts of Ruby can be removed or redefined, at will. Existing parts can be added upon. Ruby tries not to restrict the coder. This makes Ruby very flexible. [6]

Ruby has one of the largest and most well crafted packaging systems of any language today. Packages in ruby are referred to as gems. RubyGems.org hosts more than 66000 gems to date. Gems can easily be created with the gem development tools that come with ruby. Gems also have very advanced yet simple requirement specification through Gemfiles. The name of required gems and ranges of working versions are specified in the Gemfile. All required gems can then be fetched with a single command. Creating a new gem or publishing a gem are also single commands. This makes managing dependencies and reusing code in ruby a breeze. [7]

Ruby is probably one of the most prevalent languages for modern web development. PHP is most likely still the most widely used, but ruby based web projects tend to be more manageable due to the higher level frameworks and packages that are available.

Python

Python is a dynamic interpreted open source programming language. It is similar ruby in many ways. It has natural short typed syntax that is easy to read and write. For being an interpreted language, python is pretty fast. Python does not have an intuitive way of modifying existing parts of the language. [10]

Python has a packaging system that is easy to use called Python Package Index, PIP for short. PIP hosts more than 37500 packages to date and has a very simple requirement specification and installation that only requires a few commands. This makes managing dependencies with python very easy. [11]

Python is widely used for a variety of cases. In particular science and web development. It also has some mature information retrieval libraries.

Conclusion

Python and Ruby are both great programming languages that fulfill the needs for writing modifiable code that can harvest and make predictions on data.

Python is slightly faster and has more mature information retrieval libraries. Ruby has more general packages available for reuse and is somewhat more modifiable.

The speed of python does not matter in the context of a web bottle neck. The algorithms in the information retrieval libraries must most likely be reimplemented to work with a database as these tend to do all predictions in memory. Because of this and because the authors have more experience with ruby, ruby was chosen over python.

2.5 Result Testing

Root-mean-square error RMSE 2.1.1 is a way of testing estimated value against the actual value. This is how they measured the correctness of the submitted solutions in the Netflix Prize competition. To explore the correctness and improvement of the solution, this seems like a good way to go. The recommendation

system will be ran on both the Netflix dataset alone and on the Netflix dataset merged with the Twitter data.

A reduction in the RMSE value will then indicate an improvement, and indicate the gain from the collaboration of movie data and social media data.

2.6 Evaluation

This section will evaluate the various topics discussed in the prestudy.

2.6.1 Netflix

The Netflix Prize dataset is very sparse. With about 480 000 users and 17 770 movies there would have been a total of more than 8.5 billion ratings if the dataset was dense, but the dataset only contains around 100 million ratings. Algorithms that can handle this kind of sparsity must be used, such as alternating least squares and k nearest neighbors. The standard way to evaluate the correctness of a solution in these types of problems are RMSE.

2.6.2 Twitter

There are many different data types that can be retrieved from Twitter. The most dominating types and attributes are: Tweets, users, followees and followers.

There are several ways to access and retrieve this data. The REST API has the capability to retrieve the complete set of followees and followers for any given user. It lacks speed as it only allows 15 requests per 15 minute window.

The Streaming API is also great in the sense of data completeness. It has the capability of capturing all tweets for any given topic described by keywords. However, it is slow in the sense that it takes a long time to build a dense dataset because less popular films are rarely tweeted about. In the study it is shown that even running the Stream API for over a year will only produce more than one tweet for about 65% of the movies in the dataset. This is not satisfactory.

The Search API is great in the sense that it has looser limitations than the REST API. It can process a total of 180 requests per client per 15 minute window. It is however, like the Stream API limited in time span. The Search API only has a week worth of data. There is also no guarantee from Twitter's side for which time window the Search API reflects and thus no guarantee for the completeness of data. This renders the Search API almost completely similar in terms of data gathering capability to the Stream API, but inferior in certainty of data completeness. A slight advantage for Search over Stream is that keeping a persistent connection is not required which makes it easier to maintain.

Scraping is the best solution in terms of data completeness and time consumption. It achieves what the Stream API is capable of achieving in years in mere days. However, it is a more complicated and error prone procedure as the scraping must be done by the client and the content being scraped is not under its control. The scraping procedure itself is more difficult to test than to just utilize an existing API. This makes Scraping more prone to sparseness and errors in the dataset. However, the speed with which an entire site can be crawled and scraped compared to the years it takes the Stream API to collect a dataset of the same density, makes it easy to test run a scraper, correct errors as they are found and run the scraper over again.

All of the previously mentioned APIs have the advantage that all the data gathered by them can be be aggregated and analyzed. Aggregations and analysis of data obtained by these means can be legally

used to create services, including movie recommenders.

One problem with scraping is that it is not allowed by Twitter without their consent. As Twitter's consent has not been granted to scrape Twitter for the purposes of the project, a dataset based on scraping can not be gathered. The data can certainly not be used to legally create services like movie recommenders.

All of the techniques for gathering data from Twitter discussed have pros and cons in terms of how dense of a dataset they build, how likely the dataset is to contain errors, how long they take to run and the legality of how the data collection. As APIs exist for all techniques except scraping, the barrier for utilizing them are low. Implementing them and having them available has a low cost in terms of development time and each API increases the richness of available data. Implementing a scraper is also a good idea in case consent from Twitter is obtained as it has superior performance compared to any of the other techniques considered.

2.6.3 Similar systems

When it comes to recommendation systems it is important to weigh quality of result versus the time it takes to produce this result. If it is too time-consuming the result might end up being outdated. Netflix did not implement the winner solution even though the winner managed to score 10% better than the Netflix system. This was mainly because of the time it took for the predictions to be calculated and Netflix need to have a live system.

For a big dataset and low computational time k nearest neighbors has proven to be a good approach.

Chapter 3

Requirements

Contents

This chapter will describe the capturing of the requirements and go into depth on the most central ones.

3.1 Capturing the Requirements

Based on the initial task at hand and the findings in the prestudy, the requirements had a basis to get formed on. The initial task gave an overview of how the requirements of the system should look like. The prestudy on similar solutions made it clearer what was feasible to do and what would be impractical to do. For instance runtime of different recommendation algorithms and their ability to be parallelized. The prestudy on Twitter made it clear how Twitter can be explored and how it can be done in a timely manner. It also showed the difficulties in gathering sufficient data from the APIs and that scraping would work but could not be done due to legal considerations.

3.2 Functional Requirements

This section will look at the major functional requirements, and see why these are central to the system. For a full list of the functional requirements see appendix A.

FR1 The system must be able to harvest tweets and/or users from Twitter that are related to a movie in the Netflix Prize dataset

FR6 The system must be able to supplement Netflix Prize dataset with tweets and/or users from Twitter that are related to a movie in the Netflix Prize dataset

FR7 The system must be able to predict ratings of the movies for all users in any dataset that reflects the Netflix Prize dataset

FR1

Data from Twitter is essential for any recommendation to be done, so the system must be able to harvest data from Twitter. The harvested data must be related to the the movies existing in the Netflix dataset. This is because the amount of available data is vast. Unnecessary data will just produce static to the dataset, which is bad for the prediction system.

FR6

The Netflix Prize dataset consists of a set of movies and a set of users with ratings to these movies, this set is quite sparse as explored in 2.1.1. By harvesting tweets from Twitter this level of sparseness is expected to be reduced, and therefore increase the accuracy of the data. With a more complete set of data, and less sparseness, the algorithms to produce prediction of ratings will perform better [15]. So the system must be able to merge the data from the Netflix Prize and the harvested data from Twitter, so predictions can be made on the complete dataset.

FR7

The system is expected to make predictions on user ratings of movies. There will be made different datasets, at least one of the Netflix Prize dataset alone, and one where the Netflix Prize dataset is supplemented with tweets. It is important that the system is able to make predictions on the sets so comparisons and evaluations of the predictions can be made. With this comes the importance of predicting ratings of movies for all the users. It is not just one user who can get movie recommendations. The dataset will be ever growing, as new tweets will be added to the database continuously, the system must therefore be able to do prediction on the changing dataset.

3.3 Non Functional Requirements

This section will look into the major non functional requirement, and see why this is central to the system. For a full list of the non functional requirements see appendix A.

NFR1 The system must be able to cost efficiently predict a rating of a movie for a given user

NFR1

The main concern with the efficiency of the system is that if it takes too much time to calculate a prediction then the prediction might end up being of no help to the user. For instance if a user is rating a movie highly, and the rating of this movie will lead to higher certainty predictions for other movies, then the system must be able to efficiently apply this new information to the predictions. And since the dataset will be big is it important that it does predictions efficiently.

3.4 Prioritized Requirements

The order of the major functional requirements from section 3.2 is the prioritized order also. **FR1** and **FR6** must be in the order they are numbered since the next will not work without the one before, meaning, the harvested data is needed to be able to supplement the Netflix data with Twitter data.

The **NFR1** is after this, since it is important that the system has a good response time, but this will not matter without the actual data.

FR7 comes after this, since it is necessary to have a merged Netflix Twitter dataset to be able to do any collaborative filtering on the data.

Chapter 4

Design

Contents

This chapter describes the architecture and components in the architecture, how they were designed, and how they fulfill the requirements of the project.

4.1 Architecture

The architecture of the project is based on The 4+1 View Model of Software Architecture [24]. The views are used to reflect the structural aspects, temporal aspects, technological development aspects and physical aspects of the project.

The architecture uses a custom simplified version of the 4+1 notation that better fits the descriptive needs for the project.

4.1.1 Logical View

The logical view describes the components of the architecture and how they relate to one another in terms of containment, inheritance and usage. The logical view can be found in figure 4.2 and the notation for the logical view in figure 4.1

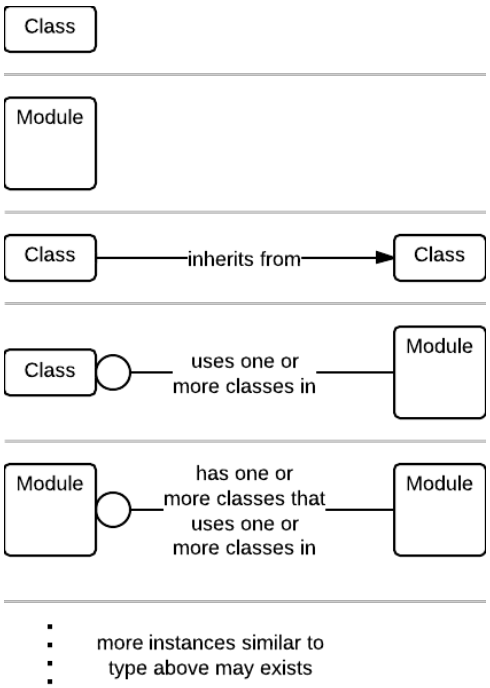

Figure 4.1: Notation for the logical view.

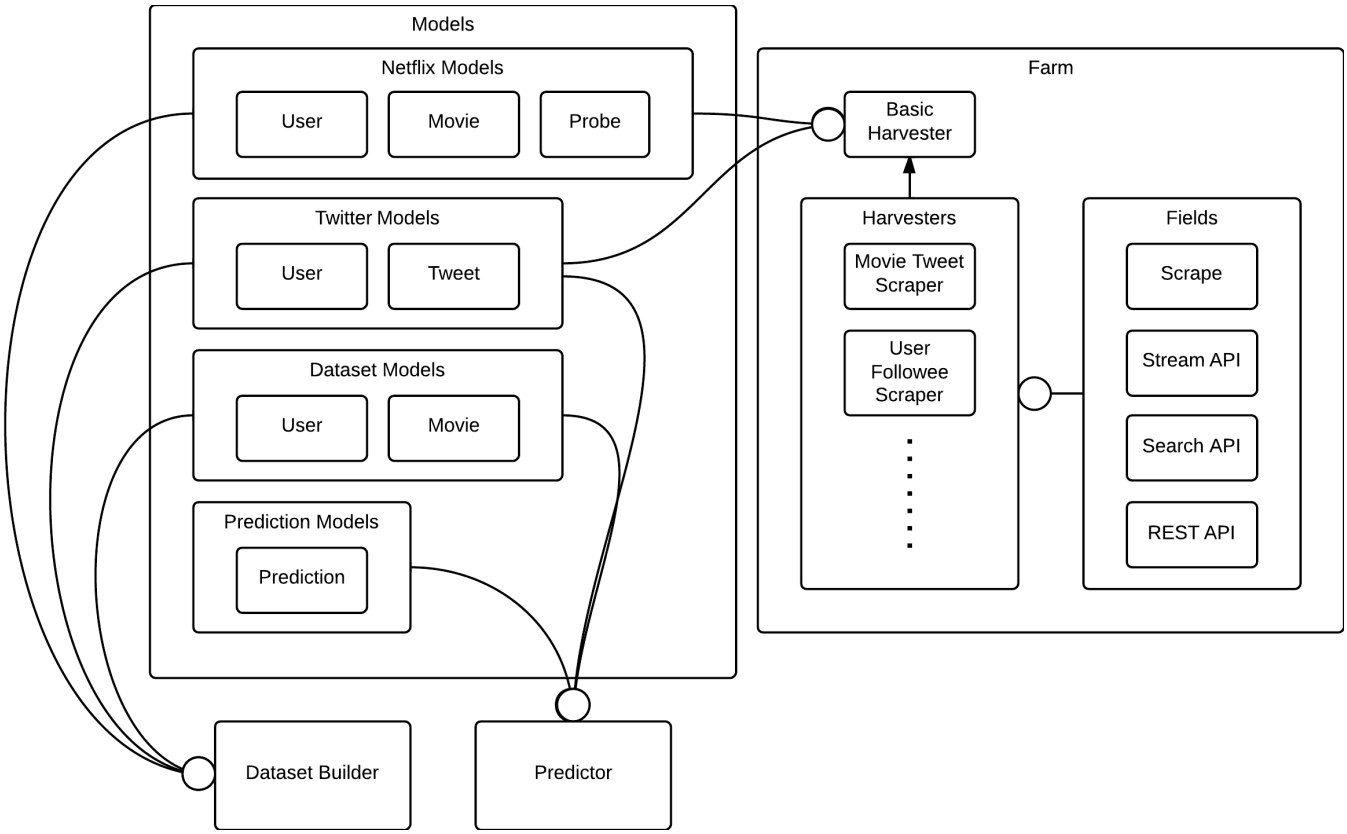

Figure 4.2: Logical view of the project. See figure 4.1 for notation.

Models

Models is the module responsible for holding the different data models that are used in the project. It is found in the upper left part of the logical view in figure 4.2. All classes in this module are mapped to and reflect the database.

The Netflix Models model the Netflix Prize dataset. Each user has a list of movie ratings and each movie has a list of user ratings. The probe is a list of movies with empty user ratings that need to be predicted.

The Twitter Models reflect the data harvested from Twitter. The schema of these models are variable as it depends on what type of data is harvested. A tweet typically contains a text and refers to a user. A user typically has a name and a user ID and may refer to tweets. As an example, it may also contain a list of followees or followers.

The Dataset Models reflects the Netflix Models after they have been mapped to fused with Twitter Models by the Dataset Builder.

The Prediction Models model what the ratings that the Predictor predicts using the Datast Models and the Probe in Netflix Models.

Farm

Farm is the module responsible for accessing and gathering data from Twitter.

The fields access Twitter's data. Most fields implement one of Twitter's existing APIs. There is also a Scrape field for accessing Twitter's web search and scraping the results. It is important to note that Twitter does not allow this without explicit consent. Each field is implemented in such a way that it delivers a subset of the Twitter data types that are outlined in the functional requirements in section 3.2. The fields combined delivers all the Twitter data types in the requirements. The reason for dividing the access over several fields is that some of the fields are unsuitable for gathering certain Twitter data types.

The harvesters uses the fields to gather and store Twitter data that matches a model that already exists in either the Netflix model or the Twitter model. It is used for finding say tweets for a movie using the scraper field, followees for a user using the rest field or users for a movie using the stream field.

Dataset Builder

The dataset builder is responsible for building a dataset that reflects the Netflix dataset using the data gathered from Twitter by the harvesters. This new dataset is stored in dataset model.

Predictor

The predictor is responsible for predicting movie ratings for a collection of users. The predictor does this by taking a test set and comparing it to several entries in a dataset. The test set is usually the probe in the Netflix models. The dataset that the test set is compared to, is either the dataset model or the Netflix model. The predictor also gradually calculates RMSE as it is predicting ratings.

4.1.2 Process View

The process view describes the actions that the components of the architecture are engaged in over time, and how their actions relate to other components. See figure 4.3

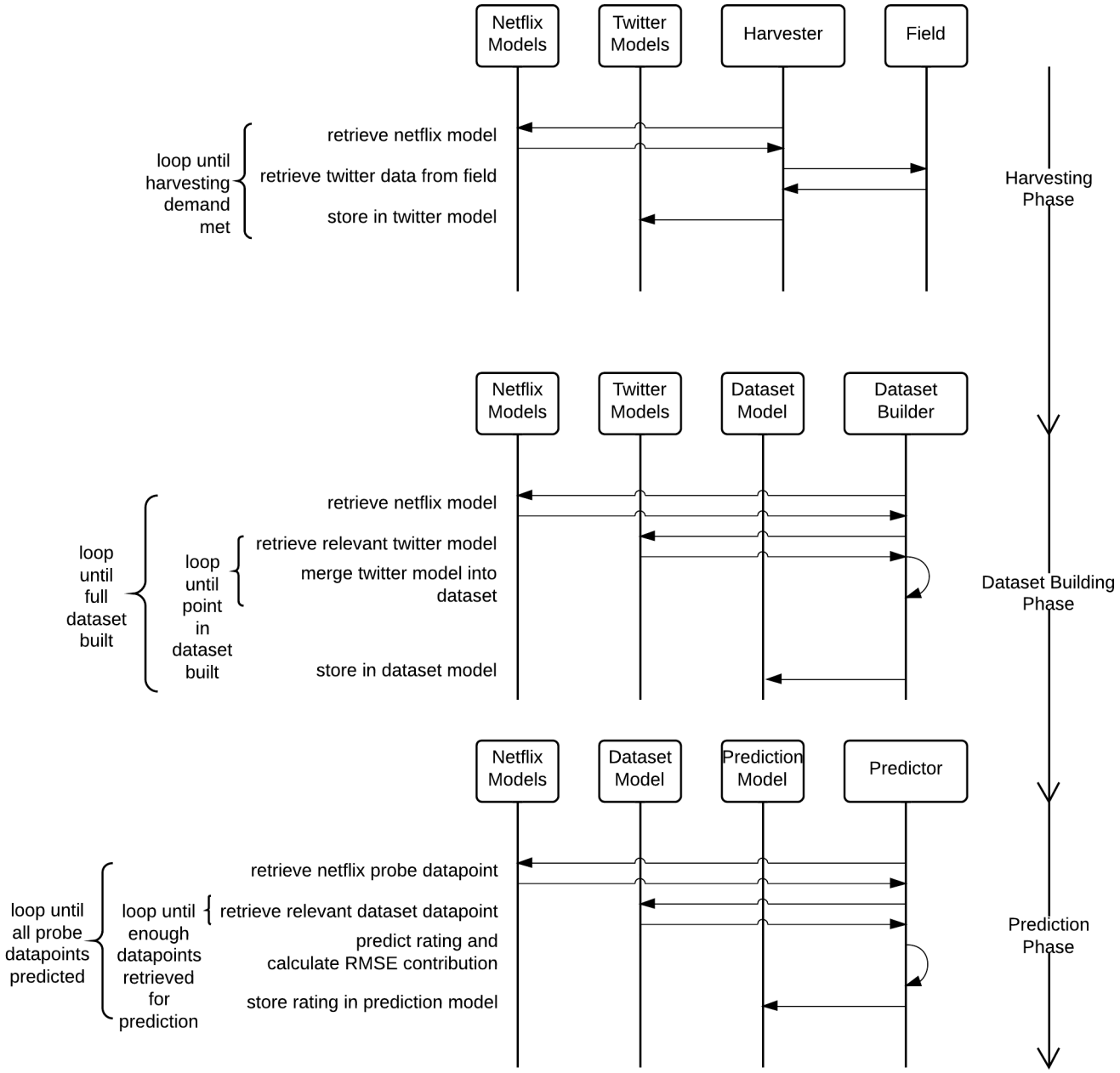

Figure 4.3: Process view showing the actions taken by various components as time passes and which components they affect. Time is moving downward. An arrow implies that data is sent from one component to another. The data being sent can be a function call, an object or any other type available to the component taking the action.

Harvest Phase

The harvest phase is responsible for gathering data from Twitter. First, the harvester retrieves an initial model from either Netflix models or Twitter models. Then, it uses one or more fields to harvest data from Twitter that is related to the initial model. This can be tweets containing the movie title in the

tweet.text, tweet.users for these tweets, users.each.followees and so forth, as outlined in the functional requirements related to harvesting in section 3.2. At last, the harvested data is stored as Twitter models.

Dataset Building Phase

The dataset building phase is responsible for building a dataset using the data gathered from Twitter in the harvest phase. First, the dataset builder retrieves a Netflix model, which is likely to be a movie. Then, it retrieves Twitter models that are relevant for the Netflix model and consolidates them into a set of data points that expresses this Netflix model. At last, the data points are stored along with the Netflix model it expresses in the dataset model. In this way, the functional requirement in section 3.2 for supplementing the Netflix Prize dataset with data from Twitter is fulfilled.

Prediction Phase

The prediction phase is responsible for using the dataset model to predict the ratings of the probe in the Netflix model and calculate the RMSE. First, the predictor retrieves a data points from the probe. Then, it gathers the relevant data points it needs from the dataset model in order to make a prediction. After this step, the rating has been predicted and the contribution to the RMSE score is calculated. The RMSE contribution is summed in the predictor and kept until all ratings have been predicted. At last, the predicted rating is stored in the prediction model. When all ratings have been predicted and all the RMSE contributions have been added to the prediction model, the RMSE can be calculated and stored. Thus, the functional requirements in section 3.2 related to giving and testing predictions are fulfilled.

4.1.3 Development View

The development view describes how the various technologies and implementations are dependent on one another. A layered approach is used to depict this in figure 4.4. It shows how the commercial off-the-shelf software (COTS) is dependent on the underlying language they are built in. Also, it shows how the modules implemented in the project are dependent on the various COTS.

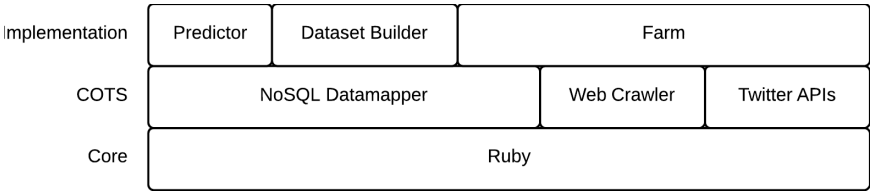

Figure 4.4: Development view showing the technology stack. Each technology block is dependent on the technology block(s) below it.

4.1.4 Physical View

The physical view describes the mapping(s) of the software onto hardware, as seen in figure 4.5. The NoSQL database communicate with the Twilm implementation. These are both stored on a single computer. This computer has Internet access. The Twilm implementation uses the Internet access to reach Twitter.

The NoSQL database can also be decoupled from the computer running Twilm and set up on several computers with replication, as shown in figure 4.6. This fulfills the system storage replication requirement in the non functional requirements in section 3.3.

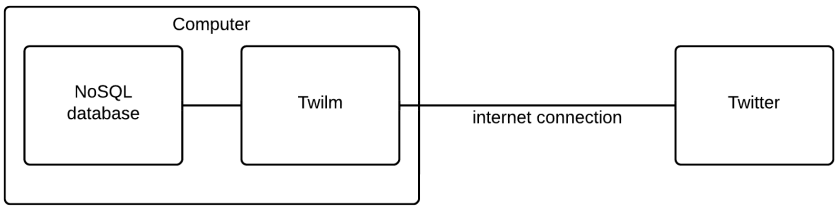

Figure 4.5: Physical view showing the components of the system mapped onto hardware. Only a single computer with Internet access to Twitter runs the entire system.

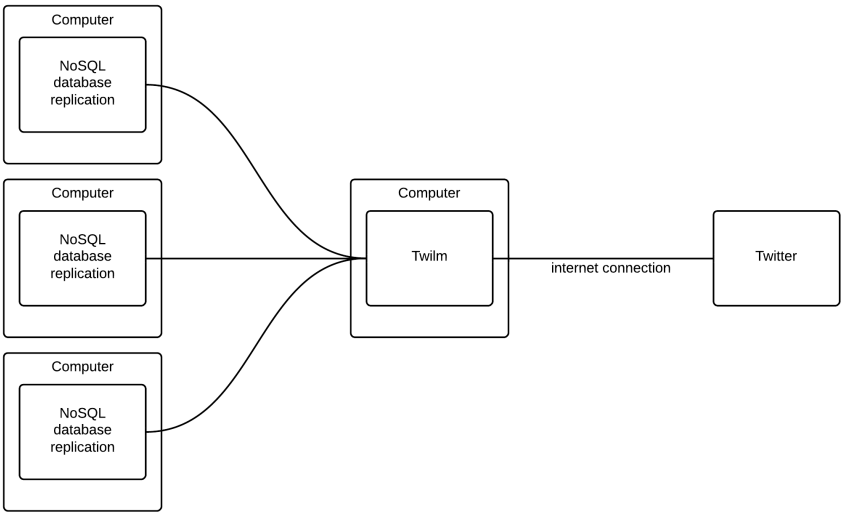

Figure 4.6: Distributed physical view showing the components of the system mapped onto hardware using database replication.

4.2 Algorithm Design

4.2.1 Dataset Building

The dataset building algorithm retrieves and iterates over a list of Netflix users in a test set. This test set is normally the probe in the Netflix Prize dataset. It then consolidates Twitter data with Netflix Prize data by building a dataset for the user based on Twitter data points related to the movies the user has rated. An illustration of this process and its beginning and resulting data structures can be seen in figure 4.7

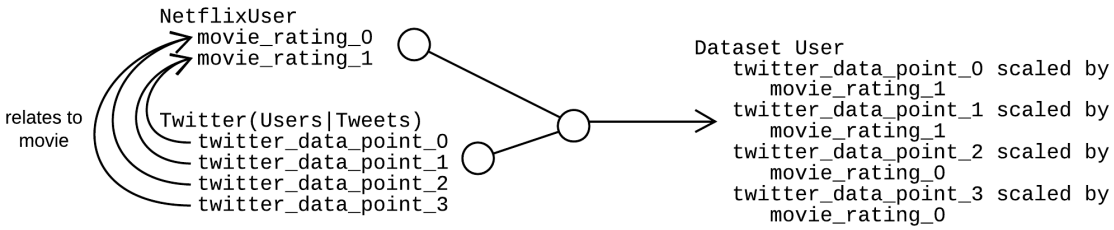

Figure 4.7: Illustration of the dataset building algorithm. The algorithm retrieves data about a Netflix user and consolidates it with the Twitter data points related to the movies the user has rated.

Retrieval

Each Netflix user that is not present in the test set is retrieved and iterated over. For each of the movies that these users have rated, the corresponding movie in the dataset that is being built containing the Twitter data points that are relevant to the movie is retrieved. If the dataset does not yet contain this movie, the relevant Twitter data points are retrieved from the Twitter model and added to the dataset movie.

Consolidating

Once Twitter data points that are related to a movie a user has rated have been retrieved, these data points must be consolidated with the user's data points. The user data points is a hash of Twitter data points that point to a number indicating whether or not the user is positively or negatively interested in this Twitter data point. Each Twitter data point related to the movie the user has rated is added to this hash and scaled by whether or not the user liked it. After Twitter data points have been added to the user data points for all movies the user has rated, they are normalized to a scale of [0, 1] for each user. The user is now expressed in the dataset entirely by Twitter data points rated from 0 to 1.

4.2.2 Prediction

As mentioned in 4.1.2 the system must produce a prediction on the dataset. As concluded in the preliminary study the most suitable algorithms to produce these predictions for this kind of dataset has proven to be a k-nearest neighbors (k-NN) approach. This is because it will allow a low response time, but yet has produced good prediction results on the Netflix Prize dataset 2.3.4.

There are many approaches to gather and select neighbors to calculate predictions with, but the approach which seems most promising is the probabilistic neighborhood selection [30], and is the approach to be taken.

The prediction algorithm design is split into 3 parts. The last part depends on if the prediction was issued because a user rated a movie, or a movie prediction rating was requested.

First - Retrieval part
The system will retrieve the data points closest to the point to predict. These points are retrieved with a k-NN algorithm, which utilizes a probabilistic selection when selecting neighbors.

Second - Evaluation part
The neighbors returned are evaluated. A prediction based on the data points is produced, and a confidence measure on this prediction is produced.

Third - Learning part

If a user rated a movie, the actual rating is compared with the predicted rating, and the data points are weighted regarding of the accuracy of the rating.

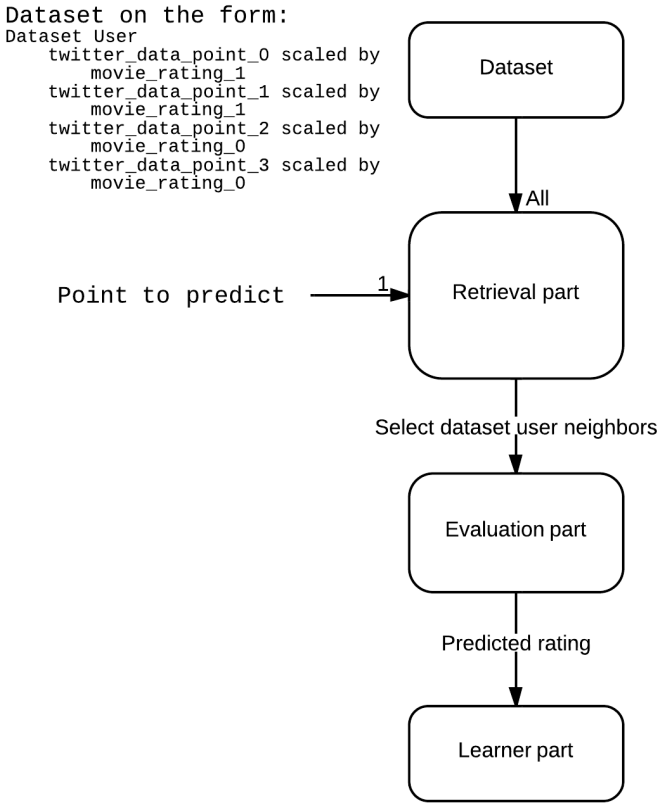

Figure 4.8: The prediction algorithm design and the parts it is separated into. The dataset consist of users on the form described in the top. With the given point the neighbors are selected from the dataset. This is then sent to the evaluation part. Here a prediction is produced.

Chapter 5

Implementation

Contents

This chapter discusses the major requirements, and implementation of the different algorithms and data structures in the system. And is meant for the reader to get insight into what was actually implemented.

5.1 Major Requirements

5.1.1 FR1

The system must be able to harvest tweets and/or users from Twitter that are related to a movie in the Netflix Prize dataset

Scrape and REST are the fields implemented. Scrape can harvest tweets and users related to a movie title. REST can retrieve users related to users that relate to tweets about movies. Stream is implemented in a side project for quick iteration and hands-on testing. It can retrieve tweets related to movie titles.

The harvesters implemented are

- NetflixMovieTweetScrape
- TwitterUserFolloweeREST

NetflixMovieTweetScrape iterates through all Netflix movies and uses the field Scrape to harvest tweets related to each Netflix movie. TwitterUserFolloweeREST iterates over all Twitter users in storage and uses the field REST to harvest followees for this user.

Thus, the system can harvest tweets and users from Twitter related to movies in the Netflix dataset and the requirement is therefore implemented.

5.1.2 FR6

The system must be able to supplement Netflix Prize dataset with tweets and/or users from Twitter that are related to a movie in the Netflix Prize dataset

Because a dataset could not be retrieved from Twitter due to time constraints on the API based fields and legal constraints on the Scrape field, there was no complete set of Twitter data points to build a dataset with. The dataset building algorithm was not implemented. Thus, this requirement was not implemented, but remains a task for future work.

5.1.3 FR7

The system must be able to predict ratings of the movies for all users in any dataset that reflects the Netflix Prize dataset

Since the data from Twitter is not gathered and combined with the Netflix data this requirement is not implemented, but rather a task for future work.

5.1.4 NFR1

The system must be able to cost efficiently predict a rating of a movie for a given user

This requirement is not implemented, but taken into account when prediction algorithm was chosen. This is as 5.1.2 and 5.1.3, a task for future work and research.

5.2 Data Structures

5.2.1 NoSQL Database Mapping

In order to achieve synchronization with mongoDB, the data mapper MongoMapper is used. MongoMapper ensures that a database record that is associated with an object is automatically updated when one of the objects attributes are changed. In the case of Twilm, these are all classes in the module Models.

The attributes are referred to as keys in MongoMapper. Instead of declaring variables in a regular ruby fashion, MongoMapper keys are declared. An example of a key used in the implementation for a TwitterUser is "key name".

MongoMappers relationship descriptors are also utilized. When an object is pointing to many instances of another object, the descriptor "many" can be used instead of declaring a regular variable. When an object is pointing to one instance of another object, the keyword "one" is used. An example used in the implementation for TwitterUser is "many twitter_tweets" and conversely, "one twitter_user" in TwitterTweet.

Validations in MongoMapper are used to ensure that certain keys and relationships are not stored with "nil" values. In the implementation, validations are used wherever there is a key that should always be stored. An example is "validates_presence_of text" in TwitterTweet.

5.2.2 Data Structure: Netflix

User

The NetflixUser data structure models the users in the Netflix Prize dataset. The implementation reflects the Netflix movies the user has rated, and also holds a cached average of these ratings. The implementation can be seen in figure 5.1.

```
NetflixUser
{
      netflix_user_id: <netflix_user_id>,
      movie_ratings: { <netflix_movie_id>: <rating> },
      average_rating = avg(<movie_ratings>.all)
}
```

Figure 5.1: The implemented data structure of a NetflixUser. Curly brackets {} denote a hash. Square brackets [] denote a list of items of a certain type. Text encapsulated in smaller than < bigger than > denotes keys in the object it is declared in or other objects. Black keys are required, while gray keys are not required.

Movie

The NetflixMovie data structure models the movies in the Netflix Prize dataset. The implementation have the keys: title and year, which reflects the title of the movie and the year it was released. It also reflects the Netflix users that have rated the movie and the rating itself. The implementation can be seen in figure 5.2.

```
NetflixMovie
{
      netflix_movie_id: <netflix_movie_id>,
      title: <title>,
      year: <year>,
      netflix_user_ratings { <netflix_user_id>: <rating> }
}
```

Figure 5.2: The implemented data structure of a NetflixMovie. Curly brackets {} denote a hash. Square brackets [] denote a list of items of a certain type. Text encapsulated in smaller than < bigger than > denotes keys in the object it is declared in or other objects. Black keys are required, while gray keys are not required.

5.2.3 Data structure: Twitter

User

The TwitterUser data structure models users harvested from Twitter. The implementation have the key name which reflects the users nickname on Twitter. It also reflects the optional many relationship to TwitterTweets which refers to a collection of Tweets posted by the user. The TwitterUser's relationship to other TwitterUsers are reflected in the optional many relationships followees and followers. The key "related_to_netflix_movie" refers to which NetflixMovie the user was harvested from and is a Twitter data point. The implementation can be seen in figure 5.3.

```
TwitterUser
{
      twitter_user_id: <twitter_user_id>,
      name: <name>
      twitter_tweets: [<twitter_tweet_id>],
      followees: [<twitter_user_id>],
      followers: [<twitter_user_id>],

      related_to_netflix_movie: <netflix_movie_id>
}
```

Figure 5.3: The implemented data structure of a TwitterUser. Curly brackets denote a hash. Square brackets denote a list of items of a certain type. Text encapsulated in smaller than < bigger than > denotes keys in the object it is declared in or other objects. Black keys are required, while gray keys are not required.

Tweet

The TwitterUser data structure models tweets harvested from Twitter. The implementation have the key text which reflects the text content of the Tweet, and the key "by_twitter_user" which reflects which user posted the tweet. The key "related_to_netflix_movie" refers to which NetflixMovie the user was harvested from and is a Twitter data point. The implementation can be seen in figure 5.4.

```
TwitterTweet
{
      by_twitter_user: <twitter_user_id>,
      text: <twitter_text>,

      related_to_netflix_movie: <netflix_movie_id>
}
```

Figure 5.4: The implemented data structure of a TwitterTweet. Curly brackets denote a hash. Square brackets [] denote a list of items of a certain type. Text encapsulated in smaller than < bigger than > denotes keys in the object it is declared in or other objects. Black keys are required, while gray keys are not required.

5.2.4 Data Structure: Dataset

This data structure is not implemented due to the fact that it is only needed by the dataset building algorithm in section 4.2.1 which is not implemented.

5.2.5 Data Structure: Prediction

This data structure is not implemented due to the fact that it is only needed by the prediction algorithm in section 4.2.2 which is not implemented.

5.3 Functional Modules

5.3.1 Fields

Scrape

Scrape is implemented using Anemone, a ruby gem that does event based crawling. Anemone is initialized with a url. It then retrieves the first page and extracts the urls from this page and append them to the crawler queue. It then initializes up to 4 threads that each grab urls from the crawler queue. It crawls the url and adds the pages to the page queue as they come in. When the last page is popped from the page queue without any new urls to crawl, Anemone finishes.

In the implementation, Anemone is initialized with a Twitter search query, like in Search Request in section 2.2.6 The primary events in Anemone that are used by the scrape is in list below:

- on_every_page
- focus_crawl
- after_crawl

The event "on_every_page" is triggered every time Anemone extracts a page from the page queue. The process passed to this event gets the page as a Nokogiri page. Nokogiri is a ruby gem for parsing XML. The knowledge of the structure of the search result page from the preliminary study in section 2.2.6 is used

together with XPATH in Nokogiri to retrieve a list of TwitterTweets from the page. The TwitterTweets are stored in mongoDB automatically with MongoMapper.

The event "focus_crawl" is triggered after every page and is responsible for delivering the list of URLs to Anemone that should be put in the crawler queue. The scroll cursor and scroll cursor URL mentioned in section 2.2.6 under Scroll Search Results is retrieved from the page and pushed to the crawler queue. If there is no scroll cursor, the Scrape prompts the harvester for another item to generate an initial search url for a new item.

The event after_crawl is triggered after no more urls can be generated. This should not happen unless the harvester has no more items that should be crawled. However, in something goes wrong, the event asks the harvester for a new item and keeps crawling.

Stream

Stream is implemented separately from the Twilm projects due to quicker development iteration. It is based on the gem tweetstream which is a full implementation of the Twitter Stream API. It connects to Twitter with OAuth for Twilm and for the personal Twitter user account of one of the authors. The OAuth details are hard coded into the application.

Stream is initialized by a harvester with a list of keywords. The keyword list retrieves all tweets that contain one or more of the keywords. The key to using Stream correctly is to define the keywords so that they return a set where the tweets wanted are a subset of this set. For Netflix movies, this could be either one of the following

- movie
- film
- (list of all release years)

Search

Search is similar but inferior to Stream, as discussed in section 2.6.2. It is therefore not implemented.

REST

REST is implemented using the ruby gem twitter. This gem is a full implementation of the Twitter REST API.

REST is contacted by a harvester to retrieve an item. The requests are functions defined in the fields class as follows.

- function followees(TwitterUser) returns [TwitterUser]
- function followers(TwitterUser) returns [TwitterUser]
- function twitter_user(twitter_user_id) returns TwitterUser
- function tweets(TwitterUser) returns [TwitterTweets]

Once an object has been retrieved from the Twitter REST API it is returned to the harvester that made the call to the function.

5.3.2 Harvesters

HarvestTwitterTweetForNetflixMoviesWithScrape

HarvestTwitterTweetForNetflixMovieWithScrape uses the field Scrape to harvest TwitterTweets related to all NetflixMovies. It iterates over NetFlixMovie.all. For the first NetflixMovie, it calls Scrape. The rest of the list is bound to a call back function "next_item_to_scrape". Scrape then knows to call this function to retrieve the next item when it is done with each item.

HarvestFolloweesForTwitterUsersWithREST

HarvestFolloweesForTwitterUsersWithREST uses the field REST to harvest followees for all TwitterUsers. It iterates over TwitterUser.all. For each TwitterUser, REST.followees(TwitterUser) is called. The list returned is merged with TwitterUser.followees and TwitterUser is saved.

5.4 Testing

5.4.1 Data Structures

The data structures were tested by creating, saving and cross checking with the database that they were being saved correctly. No formal unit testing was employed as the structures were fairly simple in structure to the point where formal test would only be testing the underlying technology. The following steps were taken to test the data structures.

- Create instances of each structures
- Save instance
- Confirm with mongoDB console

First, an instance of each data structure with all the valid fields filled out correctly was first created. For example TwitterTweet. The lack of errors confirmed that the instances were created correctly.

Then, the instance was saved using MongoMapper's document function "save!", which updates mongoDB with the document reflected in the object.

At last, the mongoDB console was opened and a query was made to make sure that the object had been created. Each field was then cross checked with the keys of the corresponding data structure instance to make sure they all matched.

5.4.2 Fields

The fields were tested by creating each field, calling the retrieval functions and making sure the retrieved items stored correctly in mongoDB. No formal functional testing was employed as the fields were fairly simple in functionality to the point where formal test would only be testing the underlying technology. The following steps were taken to test the fields.

- Create instances of each field
- Run functions available to harvester

- Confirm that a valid models returned and saved

First, an instance of each field was created in order to make sure that the fields could be created properly and connected to their APIs without complications. An example would be "rest = REST.new", which would create an instance of the field REST.

Then, all the functions in the field available to the harvester for retrieving Twitter data points were run in order to see that they were returning valid TwitterTweets or TwitterUsers. An example would be calling rest.followees(some_twitter_user) and see if it returned a list of TwitterUsers.

At last, the Twitter data points returned by each function was saved by calling "save!" on each of them. Queries for mongoDB were then constructed to retrieve these items from the database. The items returned were then cross checked to see that they matched the data in each of the Twitter data points.

5.4.3 Harvesters

The harvesters were tested by creating the harvesters and making sure that they were storing the items that they retrieved in the database. No formal functional testing was employed as the harvesters were fairly simple in functionality to the point where formal test would only be testing the underlying technology. The following steps were taken to test the harvesters.

- Create instances of harvesters
- Output each item retrieved to the console
- Confirm with mongoDB console that models stored

First, an instance of each harvester was created to make sure that there weren't any initialization errors. An example of this is "harvester = HarvestFolloweesForTwitterUsersWithREST.new"

Then, test code would print each item retrieved to the console. For the example, each item retrieved would be a list of TwitterUsers.

At last, mongoDB console queries were made to retrieve these items from the database to make sure that all objects were stored by the harvester and that the objects were the same.

5.4.4 Not tested

Testing of the RMSE score of the system is not tested since the needed data was not acquired from Twitter 6.3, neither is the merging of the Netflix Prize dataset with the Twitter-data for the same reason, which is the dataset building from the requirement in 5.1.2.

The code

Formal testing of the implemented code is not done. This is because the implemented code is of such a nature that the extra work to produce the testing code would be rendered redundant in comparison with the gain from the actual testing.

Chapter 6

Evaluation

Contents

This chapter will be used to evaluate the work and the results drawn from the research done. This chapter will also discuss why the outcome ended up as it did. This includes obstacles met during research, design and implementation, how issues could have been handled in a different way and significants of the findings.

6.1 Development Process

The development process used in the project was not modeled after any particular rigid paradigm of development, but instead followed the natural workflow of the individual programmers. Even though a particular type of predetermined workflow was not applied, there were clear tendencies in how the development happened.

The team started out with less rigid requirements. In the beginning of the project, they were something along the lines of: Store the Netflix movies in an accessible database, retrieve tweets about movies from Twitter and store them in the same database.

Each of these requirements were attacked by gathering ruby gems that seemed like they would be suitable for fulfilling the requirement. A proof of concept implementing the requirement was then created in an environment separate from other parts of the project.

As time progressed and more requirements were fulfilled the implementations of the requirements were fused into a larger project. As the project increased in complexity, and the requirements were fulfilled, more requirements emerged. They were implemented and added to the larger code base.

At one point, the code base was beginning to get complex and fusing implementations of requirements into this code base was getting difficult. This implied the need of an architecture. A basic architecture was created and the code was refactored. This process continued looping. An illustration can be seen in figure 6.1

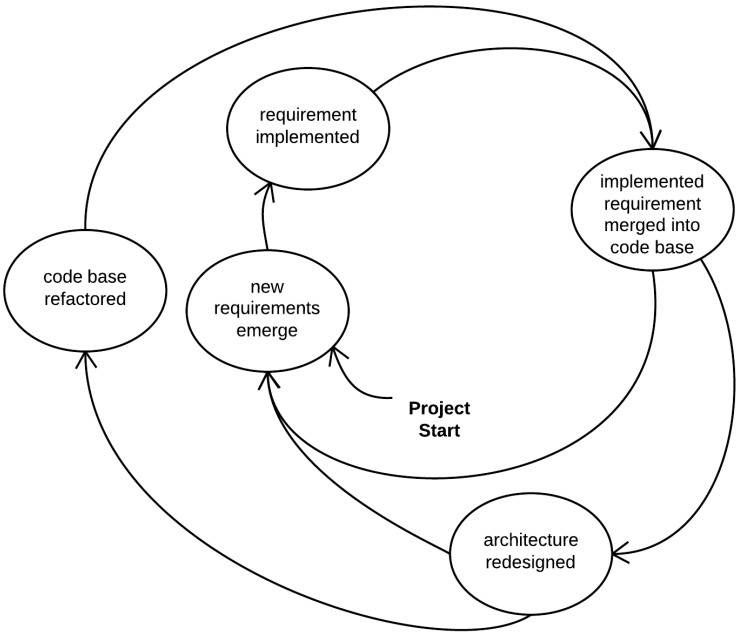

Figure 6.1: The development cycle that emerged naturally from the individual developers workflow.

Good

The approach taken was very much a learn as you go-approach. Building an architecture as a first step in the design process can be a good choice. This approach is easier if the architects have extensive knowledge of the systems involved. As this was not the case, the approach allowed the developers to

build their knowledge of the existing systems and gems as they went along. Once a knowledge base was reached, a more informed basic architecture could be created.

Doing some implementation early on also reduced the number of pointless goals. If a goal or requirement is created without knowledge of the systems involved, it is likely that the goal will not be reached or that it takes the design in a direction that does not solve other larger goals and requirements.

Bad

The project went through some major refactoring during development when the architecture was first designed. This was a difficult and time consuming task. An improvement could be made in doing this sooner, but not too soon.

Implementing the requirements outside of the code base made integration more difficult than if the requirement had been implemented within the code base in the first place.

6.2 Result Evaluation

The two types of testing used in the project were testing the assumptions in the preliminary study and one time testing of code functionality

Testing of preliminary study

The testing in the preliminary study was done by browsing Twitter for data and aggregating the results in excel or using the Google chrome developer console.

In section 2.2.5, the Stream API was tested to show what kind coverage of the movies in the Netflix Prize dataset could be expected to get by running the Stream API to harvest tweets. The test was done by manually exploring Twitter web search. This process is prone to human error. However, since the important results were statistical approximations, this type of error can be ignored. The results showed that getting coverage for the entire Netflix data set would take years. The sample size is relatively small and the results could have been better if an automated test was employed. However, the test served primarily as a loose indicator for whether or not the Stream API would serve the purpose as a main channel for harvesting tweets that would cover the dataset. A sample size of 20 assessing 3 different time spans, a month, half a year and a full year seemed sufficient enough for this. The Stream API was thus concluded to not be suitable for the project, but could complement the requirements in the future and was thus implemented, but not focused on.

In section 2.2.6, the Crawling and Scraping techniques were tested gradually as they were discovered. The tests used jQuery functions to establish that there was a correspondence between the number of tweets shown on the Twitter search result page and the number of sub elements that were to be extracted from each tweet on the Twitter search result page. The results showed correspondence even as the number of tweets on the page increased gradually. This implied that the technique for selecting the sub elements was most likely correct. This was further confirmed by running the implementation and observing that the tweets that were extracted from the page were actually the same as shown in the browser.

Testing of code functionality

The testing of code functionality is described in section 5.4. While not as solid as systematic unit and functional testing, the technique verifies at a single point in time that the different parts of the

implementation are doing what they should be doing. Though there naturally was more bug fixing, the technique served the needs of the project. The code base ended up being relatively small and thus the amount of manual testing applied was manageable. For a larger project this would not have been feasible as the workload would have increased beyond the point of being manageable

Types of testing not used

Testing the predictions with RMSE was one of the main requirements of the project, but was not implemented as predictions could not be made due to the lack of a dataset from Twitter due to constraints and limitations of the Twitter APIs and legal constrains on scraping Twitter.

6.3 Issues

Lack of Twitter dataset

A dataset that had data from Twitter could not be harvested and built for several reasons.

API Capability Issues

The APIs provided by Twitter were not sufficient to harvest data points for the 17770 movies.

The reason was mainly that the Stream API which was the best option for harvesting tweets could only deliver a coverage of 65% after running for a whole year. If each movie were to be described by 8 or more tweets, the coverage would only be 45% for this time span. See section 2.2.5.

The Search API suffered from only delivering tweets within the time span of a single week. There was no explanation of what determined this interval. It was determined that the Search API would not produce better results than the Stream API. See section 2.2.4.

The REST API showed promise in harvesting Twitter user relationships, but turned out to be limited to only 15 requests per 15 minute window, or rather one request per minute. If an average of 10 tweets were harvested per movie, it would take 123 days to harvest followers for all tweets harvested. It would take another 123 days to harvest the followees. This further complicated the retrieval of the dataset. See section 2.2.3.

Legal Issues

The scraping technique seemed the most promising. It was not formally tested as scraping is not allowed by Twitter and consent from Twitter to employ this technique could not be obtained. See section 2.2.2.

Consequential Issues

A major aspect of the project was to use data from Twitter to run predictions on the Netflix Prize Dataset. As a sufficient dataset could not be retrieved from Twitter, these predictions could not be made. The project could consequentially not meet the requirements for doing predictions and testing these predictions with their RMSE.

Chapter 7

Conclusion

Contents

This chapter will describe the final version of the product and what has been found during the project. Related work will be discussed and reviewed. And to sum it up, exploring the further development of the system and improvements which can be made to it, and more general future work.

7.1 Final Product

A lot of time was spent on the preliminary study and doing research to build a knowledge base to work from. This was to decide which direction to take amongst the vast amount possibilities. Similar solutions 2.3 was of great help, and let us understand how different approaches had been done by previous systems and what worked from them. This section will look more in depth into the status of the different parts, and conclusions made around them.

Expectations	**Fulfilled**
Understand how data from Twitter can be used for recommendations and predictions.	✓
Implement harvesting of data from Twitter.	✓
Gather data from Twitter.	✗
Supplement the Netflix Prize dataset with data from Twitter.	✗
Implement a Netflix Prize movie rating prediction system.	✗
Compare RMSE scores.	✗

Table 7.1: Main goals to be fulfilled. The fulfilled goals are marked with a ✓, and the unfulfilled are marked with a ✗

7.1.1 The fields

Done

The fields Stream, Scraper and REST were designed and implemented. Scraper could not be tested due to legal considerations.

Not Done

Search was not implemented due to the fact that it was deemed to be similar but inferior to Stream.

7.1.2 The harvesters

Done

The harvesters NetflixMovieTweetScrape and TwitterUserFolloweeREST were designed and implemented.

Not Done

TwitterUserFollowerREST was not implemented. It was useful in theory but not in practice due to the rate limit.

7.1.3 The database

Done

The choice of database was a good match to the data harvested from Twitter. MongoDB is a easily scalable database and fast, with easy replication possibilities. The data from Twitter consists of different kinds of fields, and would therefore have been more troublesome to work with in a SQL database. MongoDB, as other document oriented NoSQL databases, stores the data as JS Object Notation, which is the same notation as fetched from the Twitter APIs. There are modules for data conversion to and from JSON for any thinkable data structure.

7.1.4 The predictor

Done

Research around good prediction candidates has been made, and is and interesting topic for future work. Especially the use of k-NN for gathering similar data points to make rating predictions with.

It was interesting to see from the research done how many different approaches there are to take when suggesting user ratings, and the importance of diversity when calculating the predictions is key. There is no one model when suggesting movies to users, which will be perfect for all. Even the top two winners of the Netflix Prize competition scored better when their solutions were merged together. Even though the winners managed to produce a 10% better score than the Netflix's own system, the winners solution was not taken into production. The algorithms used to recommend at the winners level would be too time consuming to use on the complete data. It would not work well with the constant updating values.

Not Done

The prediction part was never implemented because of the missing dataset as described in 6.3.

7.2 Related Work

As was shown in the preliminary study on similar systems 2.3, there are some systems with similar characteristics as the approach taken in the project. They were of great help when exploring how to attack the issues that comes along with recommending movies, gathering information from social media and exploring big amounts of data.

Twittomender [19] had some very helpful insight. Their approach and research on how to find interesting friends, and thereby finding relevant users to harvest from, was especially helpful.

Papers on different recommendation algorithms and their produced runtime and root-mean-square error, such as [4,17,23] was of great help when deciding a suitable algorithm to use for rating predictions.

MovieTweetings 2.3.1 had an interesting take on harvesting and storing of tweets, but their approach on gathering tweets might be too strict, and would seem to reject a lot of potential tweets with the same information gain as the ones they store, but not being on the same form. They had an estimate of 500 new additions to their database daily. When Twitter is producing around 9 100 tweets a second [47], there seem to be some information lost.

7.3 Future Work

7.3.1 Twitter harvesting

The stream field is not integrated into the larger code base. This is something that would improve the quality of the Twitter harvesting framework.

The Twitter harvesting framework is sufficient for harvesting data about movies. It would be an interesting direction for further work to generalize the framework to be able to harvest supplemental data for any given dataset.

It would also be interesting to make the framework more general in the sense that new fields could easily be implemented. In this way it would be easy to supplement a dataset with data from several different sources.

Once a larger framework for data harvesting existed, more advanced distributed database systems such as Hadoop could be used to recieve and store the data. These systems have much greater capability in aggregating data than document stores and would open for much broader prediction capabilities.

7.3.2 Prediction algorithms

There is a lot of potential future work to be done regarding social media and general recommendation. In the domain of movie recommendation the exploration of different recommendation and prediction algorithms is an interesting field. The most central points regarding this work and algorithms are:

- Runtime
- Score (RMSE)
- Runtime versus score (RMSE)
- Parallelization
- Scalability

Runtime and score was just touched during this research, but opened for some interesting future work to be done, with well established testing potential through RMSE when comparing different approaches. This has opened a world of testing potential, where the main goal would be to find the most suiting algorithm/algorithms to predict ratings, which was thoroughly explored in the Netflix Prize competition, but then only on a subset of data, with few restrictions. When adding restrictions such as runtime, other solutions will emerge, these solutions could prove to be interesting to explore in depth.

When focusing in on the prediction approach taken in this research, different variants of k-NN should be explored. The different variants can be used to select neighbors. Other interesting subjects for future work includes:

- How to handle outliers in a good way regarding the data
- Best approach to value/weigh neighbors
- Exploring different heuristics for good k values.

7.3.3 Data modeling

The approach taken in this study for building the dataset with the Netflix Prize dataset and the data from Twitter can get issues with outliers, and must be research further. If the data is blindly normalized, outliers might shift data points away form their actual position, and construct a shifted illusion of their actual value.

7.3.4 Learners

The issue of comparing the actual ratings users are giving a movie with the predicted ratings, and having the system learn from these values, is an interesting issue. This would allow the system to not only depend on the harvested ratings and Netflix ratings, but also get an extra source of information when doing the prediction. As learned from the research, more information points makes for better predictions.

Future work in this field would include:

- Exploring how to weigh a correct or faulty prediction
- If a long term learning system will be of any help
- Cost of having the system learn versus the score gained
- Potential scaling issues with learning

Appendix A

Requirements

A.1 Functional Requirements

FR 1: The system must be able to harvest tweets and/or users from Twitter that are related to a movie in the Netflix dataset

FR 2: The system must be able to harvest tweet.user from Twitter

FR 3: The system must be able to harvest user.followees from Twitter

FR 4: The system must be able to harvest user.followers from Twitter

FR 5: The system must be able to harvest user.tweets from Twitter

FR 6: The system must be able to supplement Netflix dataset with tweets and/or users from Twitter that are related to a movie in the Netflix dataset

FR 7: The system must be able to predict ratings of the movies for all users in any dataset that reflects the Netflix dataset

FR 8: The system must be able to test rating predictions based on the Netflix dataset and calculate a RMSE contribution

FR 9: The system must be able to test rating predictions based on the netflix-twitter-mixed dataset and calculate a RMSE contribution

FR 10: The system must be able to track the progress of harvesting

FR 11: The system must be able to track the progress of data-building

FR 12: The system must be able to track the progress of prediction

FR 13: The system must be able to cost efficiently predict a rating of a movie for a given user

A.2 Non Functional Requirements

NFR 1: The system must be able to cost efficiently predict a rating of a movie for a given user

NFR 2: The system must be able to store all the data from neflix dataset

NFR 3: The system must be able to store all the data harvested from twitter

NFR 4: The system must be able to store all the data from predictions

NFR 5: The system storage must be able to scale

NFR 6: The system storage must have a low response time

NFR 7: The system storage must support replication for easy fault recovery

NFR 8: The system's modules must have a low coupling

Appendix B

Implementation

B.1 Implemented Functional Requirements

Implemented functional requirements are shown in green, functional requirements not implemented are show in red (strike-through).

FR 1: The system must be able to harvest tweets and/or users from Twitter that are related to a movie in the Netflix dataset

FR 2: The system must be able to harvest tweet.user from Twitter

FR 3: The system must be able to harvest user.followees from Twitter

FR 4: ~~The system must be able to harvest user.followers from Twitter~~

FR 5: The system must be able to harvest user.tweets from Twitter

FR 6: ~~The system must be able to supplement Netflix dataset with tweets and/or users from Twitter that are related to a movie in the Netflix dataset~~

FR 7: ~~The system must be able to predict ratings of the movies for all users in any dataset that reflects the Netflix dataset~~

FR 8: ~~The system must be able to test rating predictions based on the Netflix dataset and calculate a RMSE contribution~~

FR 9: ~~The system must be able to test rating predictions based on the netflix-twitter-mixed dataset and calculate a RMSE contribution~~

FR 10: ~~The system must be able to track the progress of harvesting~~

FR 11: ~~The system must be able to track the progress of data-building~~

FR 12: ~~The system must be able to track the progress of prediction~~

FR 13: ~~The system must be able to cost efficiently predict a rating of a movie for a given user~~

B.2 Implemented Non Functional Requirements

Implemented non functional requirements are shown in green, non functional requirements not implemented are show in red (strike-through).

NFR 1: ~~The system must be able to cost efficiently predict a rating of a movie for a given user~~

NFR 2: The system must be able to store all the data from neflix dataset

NFR 3: The system must be able to store all the data harvested from twitter

NFR 4: ~~The system must be able to store all the data from predictions~~

NFR 5: The system storage must be able to scale

NFR 6: The system storage must have a low response time

NFR 7: The system storage must support replication for easy fault recovery

NFR 8: The system's modules must have a low coupling

Bibliography

[1] Root-mean-square deviation. http://en.wikipedia.org/wiki/Root-mean-square_deviation, November 2013.

[2] 10gen Inc. MongoDB Introduction. "http://www.mongodb.org/display/DOCS/Introduction", 2012. [Online; accessed 2012-10-12].

[3] B. J. Jain S. Albayrak A. Said, B. Fields. User-centric evaluation of a k-furthest neighbor collaborative filtering recommender algorithm, 2013.

[4] Michael Jahrer Andreas Toscher. The bigchaos solution to the netix grand prize. http://www.netflixprize.com/assets/GrandPrize2009_BPC_BigChaos.pdf, September 2009.

[5] Hinrich Schütze Christopher D. Manning, Prabhakar Raghavan. Introduction to information retrieval. http://nlp.stanford.edu/IR-book/html/htmledition/time-complexity-and-optimality-of-knn-1.html, 2008.

[6] The Ruby Community. Ruby About. "https://www.ruby-lang.org/en/about/". [Online; accessed 2013-11-25].

[7] The Ruby Community. RubyGems.org. "https://rubygems.org/". [Online; accessed 2013-11-25].

[8] M. J. Pazzani D. Billsus. User modeling for adaptive news access, February 2000.

[9] Facebook. Facebook statistics. http://www.statisticbrain.com/facebook-statistics/, July 2013.

[10] Python Software Foundation. Python About. "http://www.python.org/about/", 2013. [Online; accessed 2013-09-13].

[11] Python Software Foundation. Python Package Index. "http://pypi.python.org/pypi", 2013. [Online; accessed 2013-09-13].

[12] The Apache Software Foundation. CouchDB about. "http://couchdb.apache.org/", 2012. [Online; accessed 2012-10-12].

[13] The Apache Software Foundation. CouchDB technical overview. "http://wiki.apache.org/couchdb/Technical%20Overview", 2012. [Online; accessed 2012-10-12].

[14] Google. Google Chrome Developer Tools. "https://developers.google.com/chrome-developer-tools/", 2013. [Online; accessed 2013-11-25].

[15] Miha Gr?ar, Dunja Mladeni?, and Marko Grobelnik. Data sparsity issues in the collaborative filtering framework. http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.106.6112.

[16] GroupLens research group. "http://www.grouplens.org", 2011.

[17] Sean Harnett. The netflix prize: Alternating least squares in mpi. `http://www.columbia.edu/~srh2144/Netflix_ALS.pdf`, March 2010.

[18] Stan Lanning James Bennett. The netflix prize. `http://www.cs.uic.edu/~liub/KDD-cup-2007/NetflixPrize-description.pdf`, August 2007.

[19] Barry Smyth John Hannon, Kevin McCarthy. Finding useful users on twitter: twittomender the followee recommender. `http://researchrepository.ucd.ie/bitstream/handle/10197/3477/fulltext.pdf?sequence=1`, April 2011.

[20] jQuery. jQuery API Reference. `"http://api.jquery.com/"`, 2013. [Online; accessed 2013-12-08].

[21] Kadence. k nearest neighbors implementation. `http://www.netflixprize.com/community/viewtopic.php?id=981`.

[22] Erik Ruggles James Sheridan Sam Tucker Kei Kubo, Peter Nelson. k nearest neighbors. `http://cs.carleton.edu/cs_comps/0910/netflixprize/final_results/knn/index.html`.

[23] Yehuda Koren. Collaborative filtering with temporal dynamics. `http://sydney.edu.au/engineering/it/~josiah/lemma/kdd-fp074-koren.pdf`, July 2009.

[24] Philippe Kruchten. The 4+1 view morm of architecture. `http://dx.doi.org/10.1109/52.469759`, November 1995.

[25] Neil Leavitt. Will nosql databases live up to their promise? `http://www.leavcom.com/pdf/NoSQL.pdf`.

[26] Jiunn Haur Lim. Predicting user ratings with matrix factorization. `http://blog.jimjh.com/static/downloads/2013/05/12/netflix.pdf`.

[27] Martin Chabbert Martin Piotte. The pragmatic theory solution to the netflix grand prize. `http://www.netflixprize.com/assets/GrandPrize2009_BPC_PragmaticTheory.pdf`, August 2009.

[28] NoSQL. List of NoSQL databases. `"http://nosql-database.org"`, 2012. [Online; accessed 2012-10-10].

[29] Sean Owen. Big, practical recommendations with alternating least squares. `http://www.slideshare.net/srowen/big-practical-recommendations-with-alternating-least-squares`.

[30] Alexander Tuzhilin Panagiotis Adamopoulos. Probabilistic neighborhood selection in collaborative filtering systems. `http://people.stern.nyu.edu/padamopo/Probabilistic%20Neighborhood%20Selection%20in%20Collaborative%20Filtering%20Systems%20-%20Working%20Paper.pdf`, 2013.

[31] Dan Pritchett. Base: An acid alternative. `http://doi.acm.org/10.1145/1394127.1394128`, May 2008.

[32] Raghu Ramakrishnan. *Database management systems*. McGraw-Hill, Boston, 2003.

[33] Chris Volinsky Robert M. Bell, Yehuda Koren. The bellkor 2008 solution to the netflix prize. `http://www.netflixprize.com/assets/ProgressPrize2008_BellKor.pdf`, 2008.

[34] W3 Schools. W3 HTML URL Encoding Reference. `"http://www.w3schools.com/tags/ref_urlencode.asp"`, 2013. [Online; accessed 2013-12-08].

[35] Luc Martens Simon Dooms, Toon De Pessemier. Movietweetings: a movie rating dataset collected from twitter. `http://crowdrec2013.noahlab.com.hk/papers/crowdrec2013_Dooms.pdf`, March 2013.

[36] Cristof Strauch. Nosql databases. http://oak.cs.ucla.edu/cs144/handouts/nosqldbs.pdf.

[37] The OptaPlanner team. Optaplanner user guide. http://docs.jboss.org/drools/release/6.0.0.Final/optaplanner-docs/html/index.html.

[38] Dimitris Tsamis Ted Hong. Use of knn for the netflix prize. http://cs229.stanford.edu/proj2006/HongTsamis-KNNForNetflix.pdf.

[39] LLC Timely Development. Timely development - netflix prize results and source code. http://www.timelydevelopment.com/demos/NetflixPrize.aspx, 2008.

[40] Tracy V. Wilson, Stephanie Crawford. How netflix works. "http://www.howstuffworks.com/netflix2.htm", May 2007.

[41] Twitter. Twitter API FAQ: Search API Timeframe. "https://dev.twitter.com/docs/faq#8650", 2013. [Online; accessed 2013-12-07].

[42] Twitter. Twitter API terms. "http://dev.twitter.com/terms/api-terms", 2013. [Online; accessed 2013-12-08].

[43] Twitter. Twitter Rate Limiting in v1.1. "https://dev.twitter.com/docs/rate-limiting/1.1", 2013. [Online; accessed 2013-12-07].

[44] Twitter. Twitter REST API v1.1 Resources. "https://dev.twitter.com/docs/api/1.1", 2013. [Online; accessed 2013-12-07].

[45] Twitter. Twitter robots.txt. "http://twitter.com/robots.txt", 2013. [Online; accessed 2013-12-08].

[46] Twitter. Twitter Terms of Service. "http://twitter.com/tos", 2013. [Online; accessed 2013-12-08].

[47] eMarketer Twitter, Huffington Post. Twitter statistics. http://www.statisticbrain.com/twitter-statistics/, May 2013.

[48] Wikipedia. Big data. "http://en.wikipedia.org/wiki/Big_data", 2012. [Online; accessed 2012-11-10].

[49] Justin Basilico Xavier Amatriain. Netflix recommendations: Beyond the 5 stars. http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html.

[50] Bo Yang. The greater collaborative filtering groupthink: Knn. http://dmnewbie.blogspot.no/2007/09/greater-collaborative-filtering.html, September 2007.