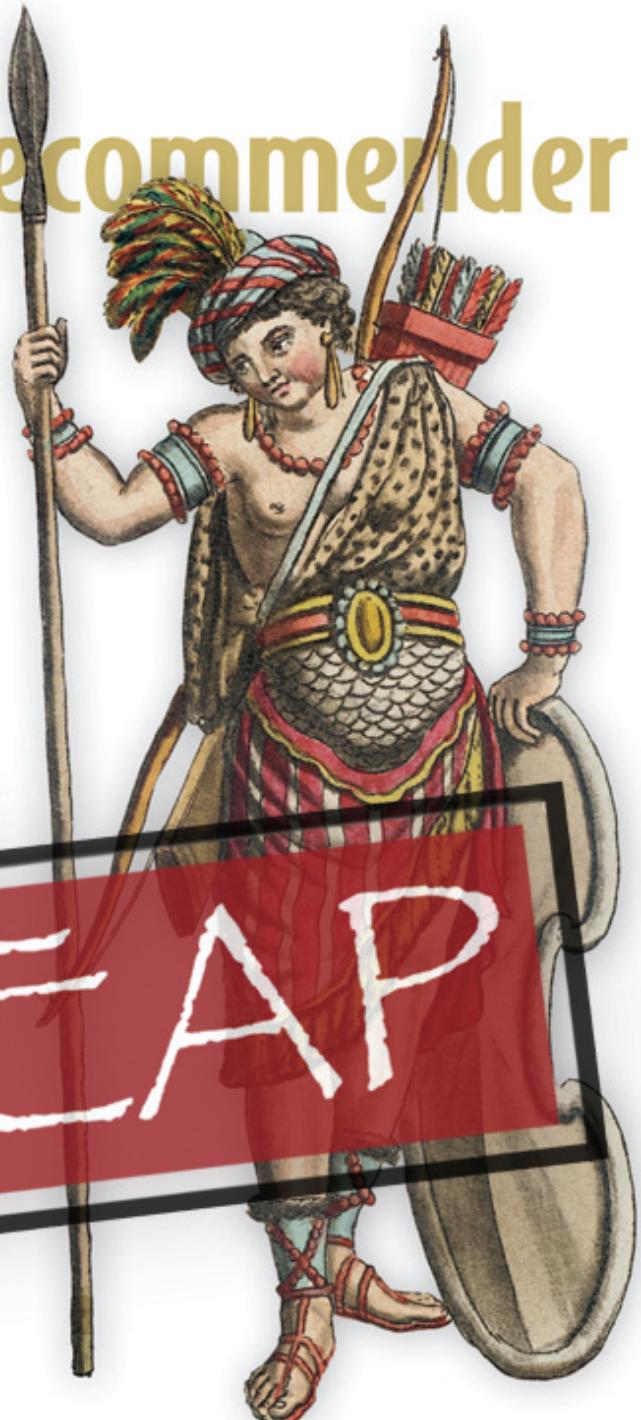


Practical Recommender Systems

Kim Falk



MANNING



**MEAP Edition
Manning Early Access Program
Practical Recommender Systems
Version 16**

Copyright 2018 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/practical-recommender-systems>

Licensed to Asif Qamar <asif@asifqamar.com>

welcome

Thank you for purchasing the MEAP edition of *Practical Recommender Systems*.

Most web applications are only a short distance from utilizing machine learning and recommender algorithms to improve their customer experience. The book will demonstrate how to implement such algorithms, with emphasis on real-world systems.

To get the most out of this book, you'll need to be comfortable with developing web applications, from deep down in databases all the way up to HTML and JavaScript.

We'll start out by introducing the basics of recommender systems – What are they made of? How do they work? How do can they guess what movie I might want to watch this afternoon?

We'll create our working environment with a movie recommendation website, generate data to work with while you learn algorithms, and explore personas and different ways to understand data. All this will be programmed Python, using Django, PostgreSQL, a bit of mathematics, and a lot of common sense.

This foundation work is necessary before we begin to explain recommendation algorithms that you will be able to implement and test on your own sandbox website. Discussion of algorithms will focus on implementation and tricks needed to make them perform. I'll provide ample examples of how these algorithms are in use existing sites and inspire the reader on how to apply this to their site.

We hope you that you enjoy *Practical Recommender Systems* and that it will occupy an important place on your digital (and physical!) bookshelf.

—Kim Falk

brief contents

1 What is a recommender?

PART 1: GETTING READY FOR RECOMMENDER SYSTEMS

2 User behavior and how to collect it

3 Monitoring the system

4 On ratings and how to calculate them

5 Non-personalized recommendations

6 The user (and content) who came in from the Cold

PART 2: RECOMMENDER ALGORITHMS

7 Finding similarities between users and between content

8 Collaborative Filtering in the Neighborhood

9 Evaluating and testing your recommender

10 Content-based filtering

11 Finding hidden genres with matrix factorization

*12 Taking the best of all algorithms – implementing hybrid
recommenders*

13 Ranking and learning to rank

14 Future of recommender systems

1

What is a Recommender

1.1 Real-life recommendations

I lived for years in Italy, in Rome. Rome is a beautiful place, with lots of food markets—not the central ones found in guide books, that are full of knock-off Gucci bags, but the ones that are outside the regular tour bus route, where farmers come and sell their products. Every Saturday we would go to see a greengrocer named Marino. We were good customers, real foodies, so he knew that if he recommended good things to us, we would buy them—even if we had strict plans to buy only what was on our list. The watermelon season was great, the many types of tomatoes offered a fountain of various flavors, and the mozzarella tasted unforgettable. Marino would also recommend for us *not* to buy something if it was not top quality, and we trusted him to give us good advice. This is an example of *recommendations*. Marino recommended the same things repeatedly which is okay with food, however that is not the case for most other types of products like books or movies.

When I was younger, before Spotify and other streaming services took over the music market, I liked to buy CDs. I would go to a music shop that catered mostly to DJs, and I would walk around and gather a stack of CDs, then find a spot at the counter with a pair of headphones, and start listening. With the CDs as context, I had long conversations with the man behind the counter. He would check which of the CDs I liked (and didn't) and recommend others based on that. I prized the fact that he remembered my preferences well enough between visits and didn't recommend the same titles to me repeatedly. This is also an example of *recommendations*.

Getting home from work, I always look in our mailbox to see if we got mail. Usually, the mailbox is full of advertisements from supermarkets, offering things that are on sale. Typically, the ads show pictures of fresh fruit on one page and dishwasher powder on the

next—all things that supermarkets like to recommend that you buy because they claim it is a good offer. *Those are not recommendations; they are advertisements.*

Once a week, the local newspaper is among the mail. The newspaper features a top-10 list of the most watched movies at the theater that week. *That is a non-personalized recommendation.*

On television, a lot of thought goes into placing commercials within the right television content. Those are *targeted commercials*. Because they expect certain type of people watching.

In February 2015, Copenhagen Airport announced that it placed 600 monitors around the airport to show commercials based on the viewer's estimated age and gender, along with information regarding the destinations at the nearby gates¹. The age and gender are inferred using cameras and an algorithm. A woman traveling to Brussels wants to see nice watches or an ad for a finance magazine, for example. A family going on vacation might be more interested in ads for sunblock or car rentals. *These are relevant commercials or highly targeted commercials.*

People usually perceive commercials on television or at the airport as a nuisance, but if we go online, the limits to what we consider invasive become a bit different. There could be many reasons for this, which is a whole topic in itself. The internet is still the Wild West, but although I think that the advertising at the Copenhagen Airport is quite invasive, I also find it irritating when I see advertisements on the net that are directed at a target group that I'm not part of. To target its commercials, websites needs to know a bit about who you are.

In this book, you'll learn about recommendations, how to collect information about the recipients of the recommendations, how to store the data, and how to use it. You can calculate recommendations in various ways, and you'll see the most used techniques. A recommender system is not just a fancy algorithm, it is also about understanding the data and your users. There is a long going discussion whether it is more important to have a super good algorithm or to have more data. Both has flipsides, super algorithms will require super hardware and lots of it. More data will create other challenges, like how to access it fast enough. Going through this book you will learn about the tradeoffs and get tools to take better decisions.

The examples above were meant to illustrate that commercials and recommendations can look similar to the user. Behind the screen the intent of the content is different, a recommendation is calculated based on what the active user likes, and what others have liked in the past, and is often requested by the receiver; a commercial is given for the benefit of the sender and is usually pushed on the receiver. The difference between the two can become blurry; in this book, we'll call everything calculated from data a *recommendation*.

¹ www.egmont.com/int/Press/news-and-press-releases/Airmagine-revolutionises-advertising-in-Copenhagen-Airport/

1.1.1 Recommender systems are at home on the internet

Recommenders are most at home on the internet because this is where we can not only address individual users but also collect behavioral data.

A website showing top-10 lists of the most-sold bread-making machines provides *non-personalized* recommendations. If a website for home sales or concert tickets shows you recommendations based on your demographics or your current location, the recommendations are *semi-personalized*. *Personalized* recommendations can be found on Amazon, where identified customers see “recommendations for you.” The idea of the personalized recommendation also arises from the idea that people are not all interested in the popular items, but also items that are not sold the most, items that are in the long tail

1.1.2 The long tail

The term *the long tail* was first coined by Chris Anderson². He identified a new business model which is frequently seen on the internet. If you don't have the limitations of a bricks-and-mortar shop, which is the limited amount of storage space, and more importantly you have a limited space to show products to users, but also limited reach since people have to come to your shop. Anderson's insight was that without these limitations you didn't have to sell only popular products, as was the usual commerce business model. This idea was long considered a losing strategy as you would need to store a lot of products that might never be sold. But if you have a webstore then you don't need to pay an expensive rent since your storage can be in places where rent is cheap, or if you sell digital content, it basically doesn't take up any space at all. The idea of long tail economy is that you can also get rich by selling lots of products, but only a few of each, and to lots of different people.

I am all for diversity, so I think it is great, with a huge catalogue then the question of how users find what they want is difficult to answer. And this is where recommender systems have their entrance. Because they help people find those diverse things that they would otherwise not know existed.

On the web, Amazon and Netflix are considered among the biggest giants both in content but also in recommendations; they're used in numerous examples throughout this book. In the following section, you'll take a closer look at Netflix as an example of a recommender system.

1.1.3 The Netflix recommender system

As you likely know, Netflix is a streaming site. Its *domain* is that of films and TV series, and it has a continuous flow of content available. The *purpose* of Netflix recommendations is to keep you interested in its content and to keep you paying the subscription month after month.

² [https://en.wikipedia.org/wiki/The_Long_Tail_\(book\)](https://en.wikipedia.org/wiki/The_Long_Tail_(book))

Netflix wants to show you content that will keep you there for as long as possible. The service runs on many platforms, so the *context* of the recommendations can differ.

Figure 1.1 shows a screenshot of Netflix from my laptop. I also access Netflix from my TV, my tablet, or even my phone. What I want to watch on each platform varies; for example, I would never watch an epic fantasy film on my phone, but I love them on TV.

Let's begin this walk-through by looking at that startup page. I have an account, and when I log into Netflix, I get the personalized start page shown in figure 1.1. The front page is constructed as a panel containing a vertical list of rows with subjects such as Top Picks, Drama, and Popular on Netflix.



Figure 1.1 The Netflix start page, before they changed the layout.

The top row is dedicated to what what's on my list. Netflix loves the list, because it indicates not only what I've watched and what I'm watching now, but also what I (at least at some point) have shown an interest in seeing. Netflix wants you to notice the following row, as that contains the *Netflix Originals* – the series which are produced by Netflix, these are important for Netflix for two reasons, they have spent a lot of money on producing them, and are in most cases only found on Netflix, and second, Netflix has to pay content owners when users

what's their content, but if that owner is Netflix then they save money. This also illustrates a point to consider, even if everything is personalized on the page, then the fact that the Netflix Originals are second row, probably isn't a result of me watching them, but rather a pursuit of an internal business goal.

CHARTS AND TRENDS

Next, we see the *Trending Now* list. Trending is a very loose term, which can mean many things, but has something to do with something that has been very popular within a very short period. The bottom row *Popular on Netflix* also has something to do with popularity, but over a longer period, maybe a week. Trends and Charts will be discussed in detail in chapter 5

RECOMMENDATIONS

The fourth row is the list of Top Picks for me and my profile. This list contains what most people would call recommendations, it shows what the Netflix recommender system predicts I'd like to watch right now. It looks almost right; I'm not into bloody, gory movies. I'd rather not see any dissections of bodies at all (which seems to be a clear sign that I've lost my connection with today's young people). Not all of the suggestions are to my liking, but I assume that it's not just my taste that Netflix uses to build this list. The rest of my household also watches content using my profile at times. Profiles are Netflix's way of letting the current user indicate who is watching.

Before introducing profiles, Netflix aimed its recommendations at a household rather than one person; it would try always to show something for mom, dad, and children³. But Netflix has since dropped that, and at least now my list doesn't include any children's show.

³ <http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html>

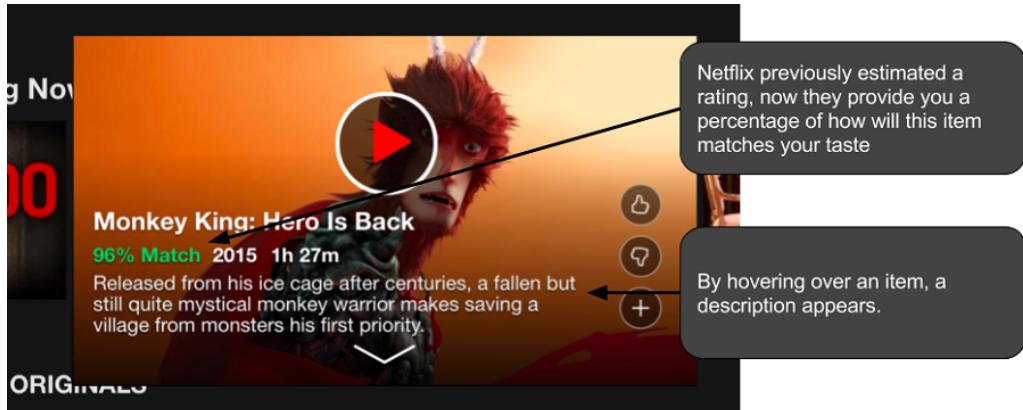


Figure 1.2 A Netflix Top Pick with predicted match

But even if Netflix is using personal profiles, I think that it is imperative to consider who is watching—not just the person with the profile, but also who else. I’ve heard rumors that other companies are working on solutions enabling you to tell the system that other people are watching too, to allow the service to deliver recommendations fitting all members of the audience, but so far, I haven’t seen any in play. Microsoft Kinect has the functionality to recognize people in front of the TV by using face/body recognition. Kinect even takes it a step further by identifying not only household members, but other people from its full catalog of users, allowing Kinect to recognize users when they’re visiting other homes.

ROWS AND SECTIONS

Back to the Top Picks of Netflix, you can find more details on the content by hovering your mouse over one of the suggestions. A tooltip appears with a description (see figure 1.2) and a predicted rating, which is what the recommender system estimates I would rate this content. You might expect that the recommendations in the “Top Picks for me” row, all have a high rating, like the one in figure 1.1, but looking through the recommendations, you can find examples of items with a low predicted rating, as shown in figure 1.3.

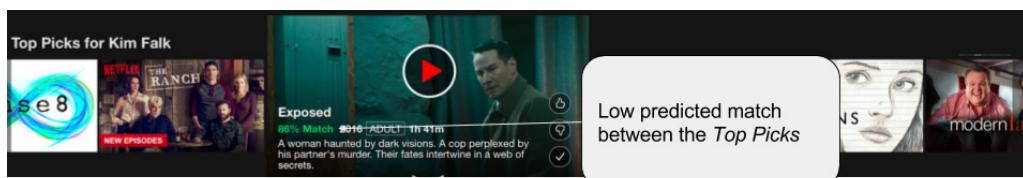


Figure 1.3 Netflix: Low predicted rating on a Top Pick

The ways of the Netflix recommender are many, so there are many possible explanations as to why Netflix chooses to recommend an item that it predicts that I wouldn't rate highly. One reason could be that Netflix is aiming for diversity over accuracy. Another reason could be that even if I wouldn't rate it maximal stars, the movie might still be something that I'm in the mood to watch now. This is also the first hint that Netflix doesn't put much value on ratings.

The titles of each row are different; some are of the type *Because you watched Suits*. These lines try to recommend things that are similar to *Suits*. Other rows are genres such as *Comedies*, which, curiously enough, contains comedies. You could say that the row titles are also a list of recommendations; you could call these *category recommendations*.

This could be the end of the story, but then you'd miss the most important part of the Netflix personalization.

RANKING

Because each of the headlines describes a set of content, this content is then ordered according to a particular recommendation system and presented to you in order of relevancy or rank, starting from the left, as illustrated in figure 1.4.

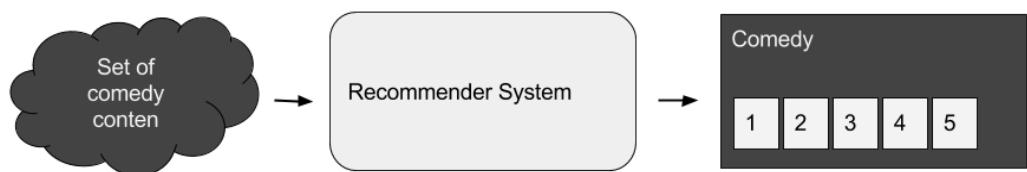
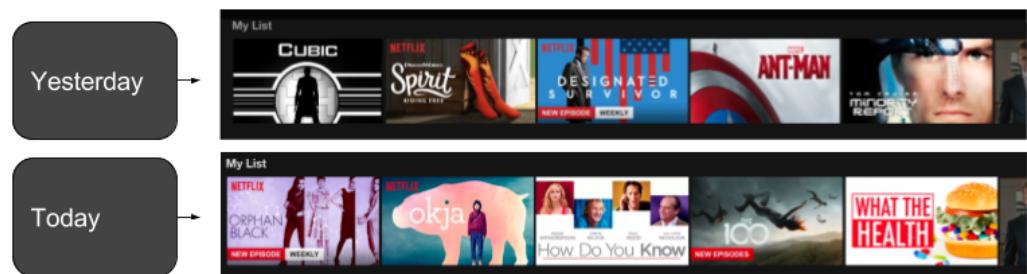


Figure 1.4: Each row is ordered by relevance.

Even in My List, which contains the content I've selected myself, the content is ordered according to the recommender system's estimate of its relevance for me now. I added the screenshot in figure 1.1 yesterday. Today my list has a new order, as shown in figure 1.5.



The Netflix recommender system will also try to recommend content that's relevant at that specific time or in a particular context. For example, Sunday mornings might be more for

cartoons and comedies, whereas evenings might be for more “serious” watching of series like *Suits*.

Another row that might be surprising is Popular on Netflix, which shows content that’s popular right now. But Netflix doesn’t say that the most popular item is the one all the way to the left. Netflix finds the set of most popular items and then orders them according to what you would consider most relevant now.

BOOSTING

A point to ponder is why Netflix adds the show *Suits* in My List, considering that I’m already watching it. Looking at figure 1.5, you can see that among the notifications, Netflix indicates that a new season of *Suits* is out; this could explain why this show shows up. *Boosting* is a way for companies to put a finger on the scale when suggestions are calculated; Netflix wants me to notice *Suits* because it is new content and therefore has a freshness value. Netflix boosts content based on freshness; *freshness* can mean that it’s new, or it has been mentioned in the news, or somebody recently wrote an excellent review. Boosting is covered in more detail in chapter 6, as it’s something that many site holders will request as soon as the system is up and running.

Please note, that there is a machine learning algorithm family called boosting⁴; what I am referring to here is something different.

SOCIAL MEDIA CONNECTION

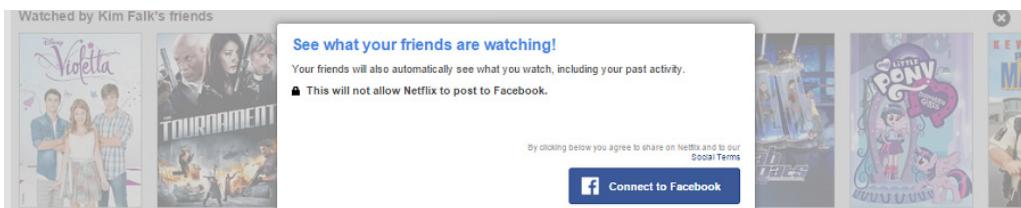


Figure 1.6 Netflix wants to know what my friends are watching

For a short period of time, Netflix also tried to use social media data⁵. If you scrolled down in your netflix page you would find something like what is shown in figure 1.6.

Netflix encouraged you to enable Facebook Connect, thereby allowing Netflix access to your friends list as well as other information. One of the advantages for Netflix is that it can find your friends and make social recommendations, based on what your friends like. Connecting with Facebook could also make watching films a much more social experience,

⁴ https://en.wikipedia.org/wiki/Boosting_%28machine_learning%29

⁵ <https://www.cnet.com/how-to/get-to-know-netflix-and-its-new-facebook-integration/>

which is something that many media companies are exploring. In this day and age, people don't sit down to watch films passively; they watch a movie while sitting with a second device (such as a tablet or a smartphone). What you're doing on the second device can have a large influence on what you watch next. Imagine that after you watch something on Netflix, a notification pops up on your phone that one of your friends liked a film, and presto, Netflix recommends that as the next thing to watch.

This social feature was however removed again somewhere between 2015-2016 with the argument that people were not happy with sharing their films with Facebook network. Which in the words of Netflix CPO Neil Hunt *is unfortunate because I think there's a lot of value in supplementing the algorithmic suggestions with personal suggestions⁶*.

TASTE PROFILE

With a page that's built almost entirely based on suggestions, it's a good idea to provide as much input as possible on your tastes.

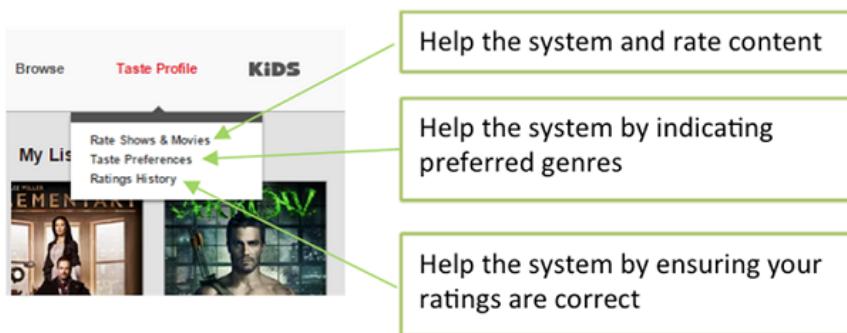


Figure 1.7 Example from how the Netflix taste profile looked in 2015

If Netflix doesn't have a clear sense of your taste, it can be hard for you to find what you want to watch. In 2016 Netflix had options which allowed users help build their taste profiles. The Taste Profile menu, shown in figure 1.7, enabled you to Rate Shows and Movies, to select genres by saying how often you feel like watching, for example, Adrenaline Rush content as illustrated in figure 1.8, or to check whether your ratings match your current opinions.

⁶ <http://www.businessinsider.com/netflix-users-dont-want-social-features-2016-2?r=US&IR=T&IR=T>

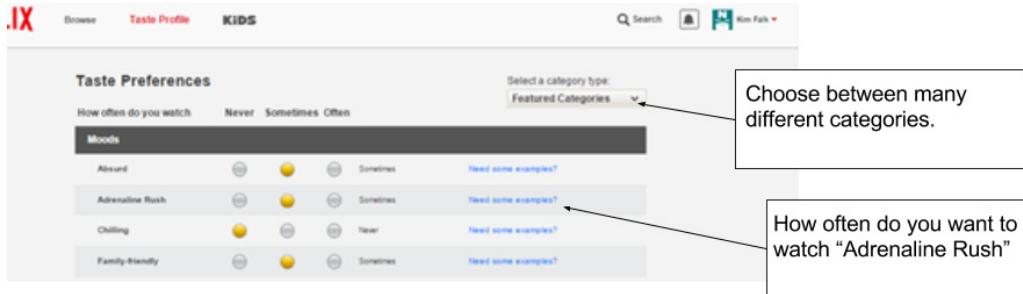


Figure 1.8 Netflix Taste Preferences

The manually inputted taste preferences enable Netflix to provide better suggestions. Asking the user for help with the taste profile is a method often used to allow the system to give suggestions for new users. But, as with so many things, there's often a difference between what users say they like and what they actually like. So, although users might provide more usage data, that explicit input is used less and less. And in fact Netflix has now removed the feature.

1.1.4 Recommender system definition

To be sure we're all on the same page, let's put down some definitions, as shown in table 1.1.

Table 1.1 Recommender system definitions

Term	Netflix example	Definition
Prediction	Netflix guesses what you will rate an item.	A prediction is an estimate of how much the user would rate/like an item.
Relevancy	All rows on the Netflix page (for example, Top Picks and Popular on Facebook) are ordered according to relevance.	An ordering of items according to what is most relevant to the user right now. Relevance is a function of the context, demographics, and (predicted) rating.
Recommendation	Top Picks for me.	The top N most relevant items.
Personalization	The row headlines in Netflix are an example of personalization.	Integrating relevancy into the presentation.
Taste profile	See figure 1.8.	A list of characterizing terms coupled with values.

With these definitions in place, we can finally define a recommender system.

DEFINITION: Recommender system

A recommender system can calculate and provide relevant content to the user, based on knowledge of the user, content, and interactions between the user and the item.

With a definition in place, you might think that you have it all figured out. But let's go through an example of how a recommendation could be calculated and how it would work. Figure 1.9 shows how Netflix might produce my Top Picks row.

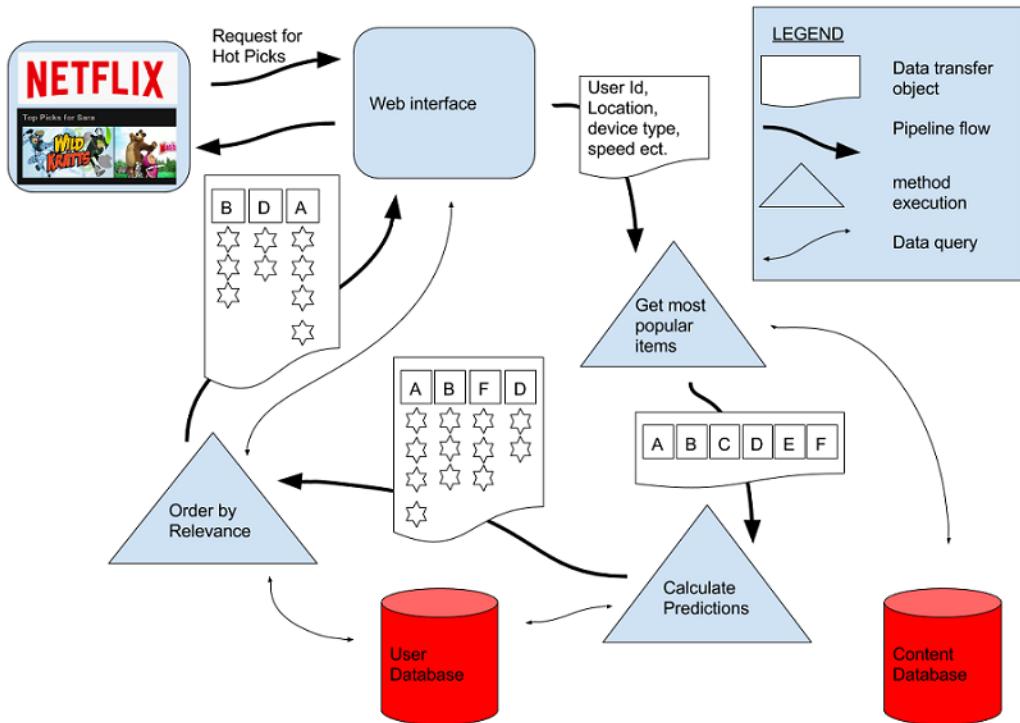


Figure 1.9: How Netflix Top Picks might be calculated

Here are the steps:

1. A request for the Top Picks list is received.
2. The server calls the recommendation system, which consists of a pipeline of methods. The first step is called `retrieve candidate items`. It Retrieves the items that is most similar to the current user's taste. from the catalog database.
3. The top five items (normally it could be 100 items or more) are piped into the next pipeline step, `calculate prediction`.

4. Prediction is calculated using the user taste retrieved from the user database. It's likely that the prediction calculation will remove one or more items from the list, due to their predicted rating being too small. In figure 1.9, two items are removed (items C and E).
5. The calculate prediction step outputs the significant items, now with a predicted rating added to them. The result is piped into an order by relevance process.
6. The order by relevance step orders the items according to the user taste, context, and demographics. It might even try to add as much diversity to the result as possible.
7. The items are now ordered by relevance. Again, item F was removed because the relevance calculations showed that one of them would not be relevant for the end user.
8. The pipeline returns the list.
9. The server returns the result.

Looking at figure 1.9, it becomes evident that there are many aspects to consider when working with recommender systems. The preceding pipeline is also missing the parts of collecting the data and building the models. Most recommender systems try to use the data shown in figure 1.10 in one way or another.



Figure 1.10 Illustrates that data can potentially be used as input data for a recommender system

This figure also illustrates another fact to take into consideration: the rating prediction is only a part of a recommendation system. Other things can also play an important role in what your system should actually display to the user. A big part of this book is about predicting ratings, it is very important, even if I made it sound like something negligible here.

1.2 Taxonomy of recommender systems

Before starting to implement a recommendation system, it's a good idea to dwell a bit on what kind of recommender system you want to roll out of the garage. A good way to start is by looking at similar systems for inspiration. In the following, you will learn a framework for studying and defining a recommender system.

In the previous section, the tour d'Netflix provided an overview of what a recommender system can do. This section explains a taxonomy to use to analyze recommenders. I first learned about it in Professor Joseph A. Konstan and Michael D. Ekstrand's Coursera course "Introduction to Recommender Systems"⁷, and have found good use for it ever since. It uses the following dimensions to describe a system: Domain, Purpose, Context, Personalization Level, Whose Opinions, Privacy and Trustworthiness, Interfaces, and Algorithms⁸

1.2.1 Domain

The *domain* is the type of content recommended. In the Netflix example, the domain is movies and TV series, but it can be anything: sequences of content such as playlists, best ways to take e-learning courses to achieve a goal, job listings, books, cars, groceries, holidays, destinations, or even people to date. The domain is significant because it provides hints on what you would do with the recommendations.

The domain is also important because it will indicate how bad it is to be wrong, if you are doing a music recommender then it is not really that bad if you recommend music which is not spot on. While if you are recommending foster parents to children in need, then the cost of failure is quite high.

The domain will also dictate if you can recommend the same thing more than once.

1.2.2 Purpose

What is the aim of the site, both for the end user and for the provider? For end-users, the use of Netflix recommendations is to find relevant content that they want to watch at that specific time. It might sound silly, but imagine that you didn't have any ordering or filtering. How would you ever find anything in the Netflix catalog when it has more than 10,000 items?

The purpose for the provider (in this case, Netflix) is ultimately to make customers pay for the subscription month after month by providing content they want to watch, right at their fingertips.

Netflix considers the amount of content viewed as a deciding factor in how they're doing. Often measuring something else instead of your direct goal is called using a proxy goal. Using a proxy goal is something you should be very careful about because it can inadvertently end

⁷ <https://www.coursera.org/learn/recommender-systems-introduction/>

⁸ The taxonomy first appeared in the book *Word of Mouse* by John Riel and Joseph A. Konstan.

up measuring other effects than what you wanted, more time spent on the Netflix platform could also mean very frustrated customers that searches and searches, or because the site keeps stalling while playing something⁹. Behind the scenes, there might also be considerations to balance things in such a way that Netflix pays the least money possible for what you are watching. Netflix probably pays less to offer 10-year-old episodes of *Friends* than a newer series or, even better, a Netflix original series; then they don't have to pay a license to anybody.

A purpose could also be to give information or to help or educate the user. In most cases, however, the purpose is probably to sell more.

What type of customers do you want to serve: consumers who arrive once and expect good recommendations, or loyal visitors who create profiles and return on a regular basis? Will the site be based on automatic consumption (for example, the Spotify radio station, which keeps playing music based on a song or an artist)?

1.2.3 Context

The *context* is the environment in which the consumer receives a recommendation. For example, Netflix delivers content on many platforms. The device the customer is using is the context. The context is also about the current location of the receiver, what time it is, and what the receiver is doing. Does the user have time to study the suggestions or is a quick decision needed? The context could also be the weather around the user or even the users mood! Consider a search for a café on Google Maps. Is the user sitting at a computer in the office and looking for a good coffee bar or is he standing on the street just as it starts to rain? In the first scenario, the best response would identify good-quality cafes in a bigger radius, whereas in the second scenario, recommendations would ideally contain only the nearest place to drink coffee while the rain passes. Foursquare is an example of an app where you can find cafes. We will look at Foursquare in chapter 12.

1.2.4 Personalization level

Recommendations can come at many levels of personalization, from using basic statistics to looking at individual user data. Figure 1.11 illustrates various levels of personalization.

⁹ I recommend reading *Weapons of Math Destruction* by Cathy O'Neil if you want to know more about how wrong it can go when you use proxy goals.

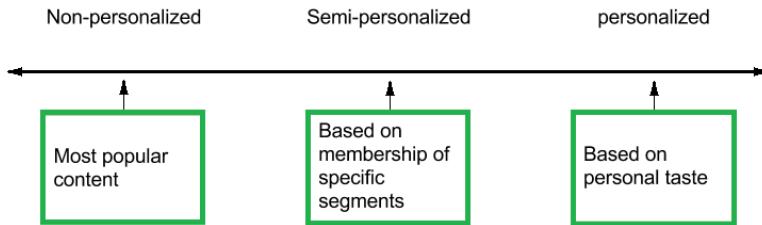


Figure 1.11 Personalization levels

NON-PERSONALIZED

A list of the most popular items is considered a non-personalized recommendation: the chances are that the current user might like the same items as most others do. Non-personalized recommendations also include showing things ordered by date, such as showing the newest items first. Everybody who interacts with the recommender system receives the same list of recommendations. Non-personalized recommendations also include when a café suggests drinks Friday afternoon, cappuccinos in the morning, but brunch on weekend mornings.

SEMI/SEGMENT-PERSONALIZED

The next level of personalized recommendations divides users into groups. You can segment groups of users in many ways: by age, by nationality, or by distinct patterns such as business people or students, car drivers or bicycle riders.

A system selling concert tickets, for example, would recommend shows based on the user's country or city. If a user is listening to music on a smartphone, the system might try to deduce whether that person is exercising, by using the GPS and seeing whether the device is moving. If it is stationary and at home, the consumer is probably sitting on a sofa and the appropriate music might be different. The recommender system doesn't know anything personal about you as a person, only as a member of a group. Other people who fit into the same group will get the same recommendations.

PERSONALIZED

A personalized recommendation is based on data about the current user that indicates how the user has interacted with the system before. This generates recommendations specifically for this user. Most recommender systems also use segments and popularity when creating personalized recommendations. Amazon's *Recommended For You* list is personalized. Netflix is an extreme example of personalized recommendations.

Usually, a site applies various types of recommendations. So far, only a few examples such as Netflix offer everything personalized. On Amazon, you will also find Most Sold Items, which is nonpersonalized, as well as the *Customers Who Bought This Also Bought This* list, which

provides seeded recommendations. These are recommendations based on a *seed*, which is the current item.

1.2.5 Whose opinions

Expert recommenders are manual systems where experts recommend good wines, books or similar. These systems are used in areas where it's generally accepted that you need to be an expert to understand what is good. I would say that the days of expert websites are mostly over, and so the Whose Opinions parameter isn't used much nowadays. Almost all sites use the opinions of the masses.

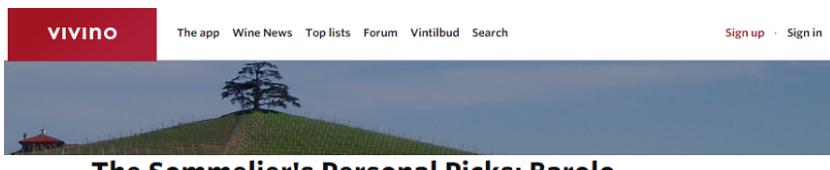


Figure 1.12 Vivino.com provides expert wine recommendations (the recommendations were cut away to save space).

They say that there's no rule without exception, and a few expert sites are still left. An example is the sommelier's recommendation on the wine site called vivino.com, shown in figure 1.12. In the first quarter of 2017, Vivino announced that they will add a recommender system to their app¹⁰ to help users find new wines to taste based on their rating history.

1.2.6 Privacy and trustworthiness

How well does the system protect users' privacy? How is the collected information used? In Europe, it's common to pay money into a pension, which is handled by a bank. Often these banks offer different kinds of retirement savings plans. A system that recommends these should have strict rules for privacy. Imagine filling in an application for a retirement savings plan and describing that you have back problems, and a minute later receiving a phone call from a chiropractor with great offers to handle your exact problem. Or even worse, you buy a special bed for people with back problems, and an hour later, you receive an email that your health insurance premium has gone up.

Many people consider recommendations as a form of manipulation because they present choices that customers are more likely to pick than if they were offered just a random

¹⁰ <http://www.digitaltrends.com/home/vivino-market-wine-recommendations/>

selection. And most shops are trying to sell more, so the fact that stores that use recommendations sell more makes people think that they are being manipulated. But if that means watching a film that would entertain rather than bore, then I would say it is okay. So, manipulation is more about the *motive* for showing a particular item rather than the *act* of showing it. If you're recommended inappropriate and non-optimal medicine because the vendor buys the website owner better dinners, then that is manipulation, which is to be frowned upon.

When the recommendation system starts performing, and an increase in business can be measured, many will find it tempting to inject vendor preferences, overstocked items, or maybe preferences for which brand of pills customers buy. However, beware: if customers start feeling manipulated, they'll stop trusting the recommendations and eventually find them somewhere else.

The moment that recommendations have the power to influence decisions, they become a target for spammers, scammers, and other people with less-than-noble motives for influencing our decisions.

—Daniel Tunkelang¹¹

Trustworthiness indicates how much the consumer will trust recommendations instead of considering them as commercials or attempts at manipulation. In the Netflix example, I talked about how predictions can be discouraging for users if the estimated prediction is far off from the user's actually rating. This is about trustworthiness. If the user takes the suggestions seriously, the system is trustworthy.

1.2.7 Interface

The *interface* of a recommender system depicts the kind of input and output it produces. Let's have a look at each

INPUT

Netflix once enabled users to enter taste preferences by rating content and adding preferences on genres and topics, this can be used as input to a recommender system.

The example described with Netflix is an explicit input. Where you the consumer manually add information about what you like. Another form of input is implicit, meaning that the system tries to deduce taste by looking at how the user interacts with the system. Feedback will be handled in more detail in chapter 4.

¹¹ www.linkedin.com/pulse/taste-trust-daniel-tunkelang

OUTPUT

Netflix outputs recommendations in many ways. Netflix estimates predictions, provides personalized suggestions, and shows popular items (which normally is in the form of a top-10 list, but Netflix even personalizes that).

Types of output could be predictions, recommendations, or filtering. If the recommendations are a natural part of the page, it's called an *organic presentation*. The rows shown on Netflix are an example of organic recommendations; Netflix doesn't indicate that these are recommendations; they're just an integral part of the site.

Although the examples illustrated in figure 1.13 are *nonorganic*, Hot Network Questions explicitly states what it is (non-personalized recommendations). Amazon shows nonorganic personalized recommendations in its Recommended for You list, and the *New York Times* has nonorganic recommendations showing the most emailed articles.



Figure 1.13 Examples of nonorganic, non-personalized recommendations: Hot Network Questions from Cross Validated, Recommended for You at Amazon and Most Emailed from the New York Times.

Some systems explain the recommendations. Recommenders with that ability are called *white-box* recommenders, and recommenders that don't are called *black-box* recommenders. Examples of each are shown in figure 1.14. This is important to consider when choosing an algorithm because not all of them provide a clear path back to the reasons for a prediction.

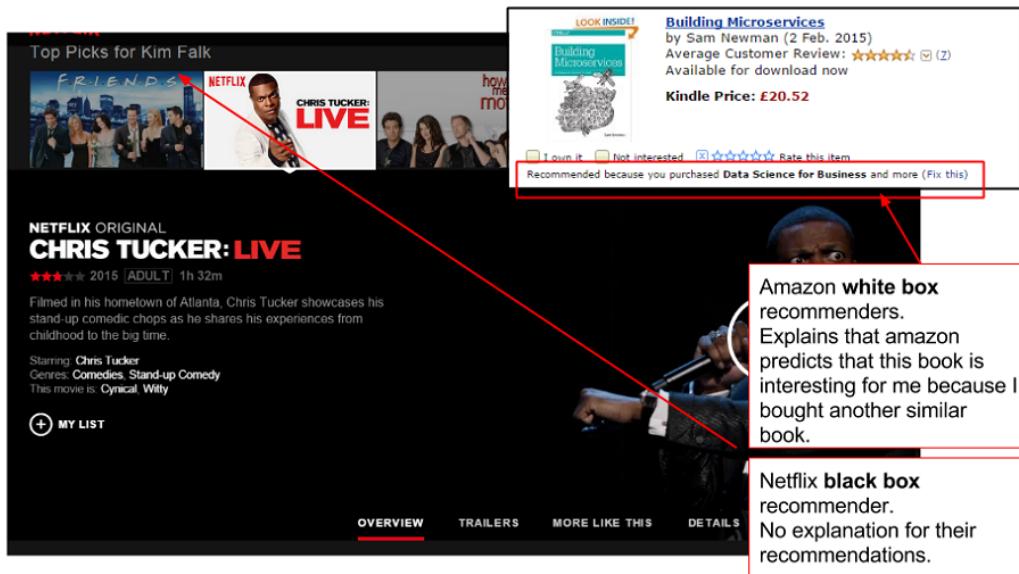


Figure 1.14 Black box and white box recommendations

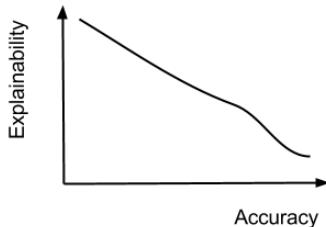


Figure 1.15 Explainability vs. quality of recommendations

Deciding whether you want to produce a white-box or black-box recommender is important because it can put constraints on which algorithms you can use. The more your system needs to explain, the simpler the algorithm will have to be. Often you can consider the decision as shown in figure 1.15. The better the quality of the recommendations is, the more complex and, therefore, the harder to show explanations. The problem is also known as *model accuracy - model interpretation trade-off*.

I have worked on a project requiring both explainability and quality to be optimum. To solve this, we had to build another algorithm on top of our recommender system to allow for the good-quality recommendations while also having a system that connected the evidence with the result.

Recommender systems have become extremely common in recent years, so there are many examples out there to look at. Often they are implemented in relation to movies, music, books, news, research articles, and products in general. But recommender systems also have a place in many other regions such as financial services, life insurance, online data, job searches, and in fact everywhere there are choices to be made. This book primarily uses websites as examples, but there is no reason for not working in other scenarios.

1.2.8 Algorithms

Various algorithms will be presented in this book. There are essentially two groups, and they depend on the type of data you use to make your recommendations. Algorithms that use usage data are called *collaborative filtering*. Algorithms that use content metadata and user profiles to calculate recommendations are called *content-based filtering*. There is a third type which can be a mix of the two types, called hybrid recommenders.

COLLABORATIVE FILTERING

Figure 1.16 shows a diagram illustrating one way of doing *collaborative filtering*. The outer set is the full catalog. The middle set is a group of users who have consumed similar items. A recommender system recommends items from the purple set, assuming that if users liked the same things like the current user, then the current user will also like other items this group has consumed. The group is identified by having an overlap between what the individual users have liked and what current user liked. Then the gap of content, which the current user is missing, will be recommended (the part of the middle circle that isn't covered by the circle representing the current user's likes).

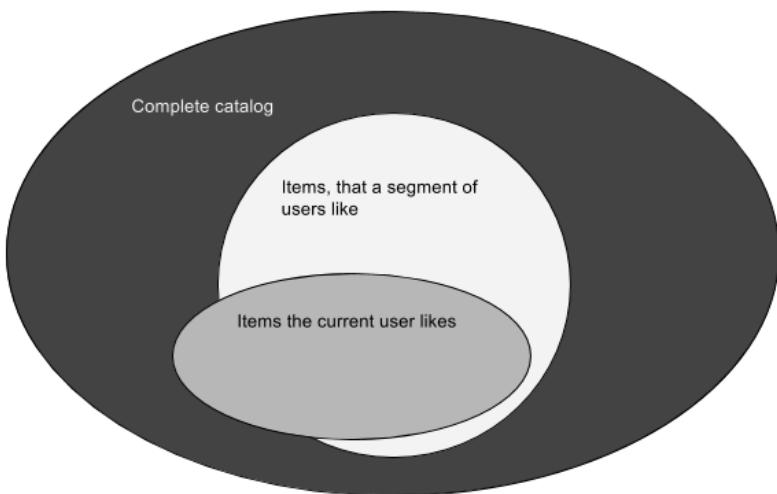


Figure 1.16 Collaborative filtering diagram

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/practical-recommender-systems>

Licensed to Asif Qamar <asif@asifqamar.com>

There are many ways to calculate collaborative filtering recommendations; you'll see a simple version in chapter 8 and again in chapter 11, where we talk about matrix factorization algorithms.

CONTENT-BASED FILTERING

Content-based filtering is about using the metadata you have on the items in your catalogue. In context of Netflix which is movies we would use descriptions of the movies. Depending on the specific algorithm used, the system can either calculate recommendations using either be done by taking the items the users has liked and find similar content to ,alternatively between the items and user profiles. Or simply between items. The system calculates a profile for each user, which contains categories of the content. If Netflix used content-based filtering, it would create a user profile comprising genres such as thrillers, comedies, drama, and new films, and give values to them all. Then a film gets recommended if it has similar values as the user. Here's an example:

User Thomas likes *Guardians of the Galaxy*, *Interstellar*, and *Game of Thrones*. Each film is rated according to a five-point system. The following could be a way of looking at the three films:

	Sci-Fi	Adventure
Interstellar	3	3
Game of Thrones	1	5
Guardians of the Galaxy	5	4

Based on this information, you build a profile of Thomas indicating Sci-Fi: 3, Adventure: 4. To find other films to recommend, you look through the catalog to find films similar to Thomas' profile.

HYBRID RECOMMENDER

Both collaborative filtering and content-based filtering have strengths and weaknesses, collaborative filtering needs to have a lot of feedback from the users to work probably, while the content-based filtering needs good descriptions of the items, to work. Often recommendations are produced as a mix of the output from the two types of algorithms we talk about above, plus other types of input, which could be as distance from the place, time of day.

1.3 Machine learning and the Netflix Prize

A recommender system is about predicting what content a user needs right now. There are many ways to predict this. Building recommender systems has become a multidisciplinary sport that takes advantage of computer science fields such as machine learning, data mining and information retrieval, and even human-computer interaction. Machine-learning and data-

mining methods enable the computer to learn to predictions by studying examples of what it should predict. Consequently, recommendations can be constructed by using these prediction functions. Many recommender systems are centered around various machine-learning algorithms to predict user ratings of items, or to learn how to correctly rank items for a user.

One reason that the field of machine learning is growing is that people are trying to solve the recommender system problem. The aim is to implement algorithms that will enable computers to suggest our secret wishes, even before we know them ourselves.

Many will claim that the catalyst for this interest in applying machine learning to recommender systems was the famous Netflix Prize. The Netflix Prize was a competition hosted by Netflix that offered one million dollars to anyone who could come up with an algorithm that improve their recommendations by 10%. The competition began in 2006, and it took almost three years for somebody to win it. Many people, universities, and companies tried to do it. In the end, it was a hybrid algorithm that won. A hybrid algorithm runs several algorithms and then returns a combined result from all of them. (we will talk about hybrids in chapter 11).

Netflix never used the winning algorithm. The biggest reason was that it was such a complicated algorithm that the performance hit on the system could not justify the improvements.

Sadly, we don't have Netflix to play with while learning about recommender systems. Instead, I have implemented a small demo site called MovieGEEKs to show off the things described in the book. The site would require a lot of tweaking before it would be production ready. Understanding recommender systems is its key purpose.

1.4 The Movie GEEKs website

This book is about how to implement recommender systems. It will provide you with the tools to do that, no matter which platform you want to use for your recommendation system.

However, to do anything interesting with a recommender system you need data and to get a feel for how it is working its not enough just to look at numbers. This book focuses on websites, but that doesn't mean that everything written here doesn't apply to any other type of system. The following is a short introduction to the framework in which we'll do our dance.

We will use a website built using a Django Website. And I encourage you to download and use the site as you read through the book, as it will help you understand what is going on. The fact that it is a Django site or something else is not so important; I will point you to where to look.

Django website and framework

If the words *Django webframework* sounds strange to you, please have a look at the Django documentation at <https://www.djangoproject.com/start/overview/>.



The website is downloaded once and will contain all the functionality described in this book. The scenario we will pretend we are doing is as follows:

Imagine that you have a customer who wants to take his DVD selling online. I imagine an old DVD rental shop that's in Bath, in the United Kingdom, with an owner who wishes to try movie selling on the internet.

(The store sadly doesn't exist any longer.¹² The shop was anything but electronic; it was all managed with small paper cards, and although you might think that sounds impossible, it all seemed to work fine. In real life, I don't think the owner would ever have taken his business online, but one of the unique things about this place is that you'd always get superb recommendations; the owner would do a monthly review—expert opinion recommendations—and the people who worked there knew everything there is to know about films.) I like to think of recommender systems as an attempt to give personal service to people on the net. The following is a brief description of what the fictive owner wants.

1.4.1 Design and specification

To get started on the project, you'll need to put down some overall points for the design.

The main page of the site should show visitors the following:

- A tiled area of movies
- An overview of each film, without leaving the page
- Recommendations as personal as possible
- A menu list containing the genres

Each movie should have its own page containing details as follows:

- Movie poster
- Description
- Rating

Each category should have a page containing the following:

- Same structure as the front page
- Recommendations specific to the category

¹² A sad day for movies as Bath rental shop On the Video Front shuts: www.bathchronicle.co.uk/sad-day-movies-Bath-rental-shop-Video-shuts/story-17883596-detail/story.html

1.4.2 Architecture

You'll use Python and the Django web framework to implement this site. Django enables you to split a project into different applications. Figure 1.17 shows a high-level architecture and provides an illustration of which applications will build the site.

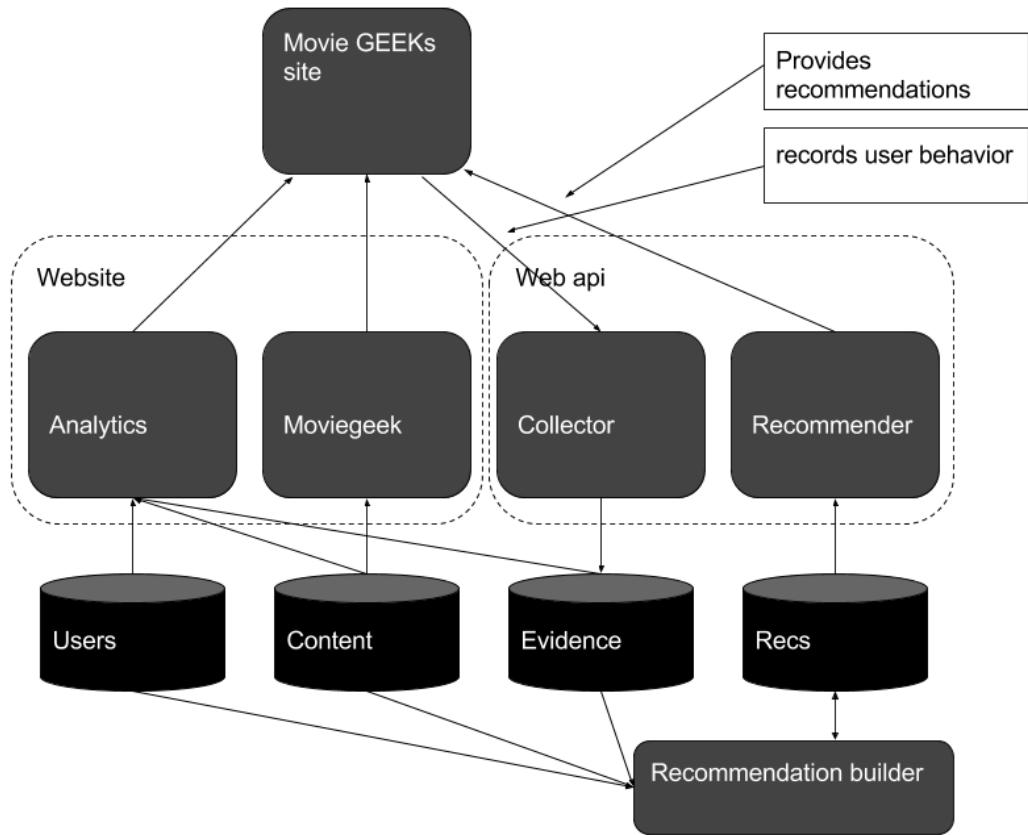


Figure 1.17 MovieGEEKs architecture

Let's do a quick walk-through:

- *Moviegeek*—This is the main part of the site. Here the client logic (HTML, CSS, JavaScript) is placed along with the Python code responsible for retrieving the movie data.
- *Analytics* —The ship's bridge, where everything can be monitored. This part will use data from all the databases. The analytics part is described in chapter 4.
- *Collector*—This handles the tracking of the user behavior, and stores it in the evidence

database. The evidence logger is described in chapter 2.

- Recs—This is the heart of this story, and is what will add the edge to this site. It will deliver the recommendations to the MovieGEEKs site. The recs part is described in chapter 5 and the rest of the book.
- Recommendation builder—This will pre-calculate recommendations, which the recs will use to provide elaborate recommendations to the user. You will meet the recommendation builder for the first time in chapter 7.

Each of these components/applications will have exciting data models and features that hopefully will blow future visitors away. But not to spoil everything now; please find details in each of the respective chapters.

The website is called Movie GEEKs and is a movie site mainly for the single reason that a dataset is available that contains a long list of content—movies, users, and ratings. Even more important, the content includes URLs that translate into movie posters, which makes working with it much more fun.

Let's look at a screenshot. Figure 1.18 shows the front page, which shows the landing page. When the user clicks a movie, a pop-up appears, giving more information and a link to even more details.

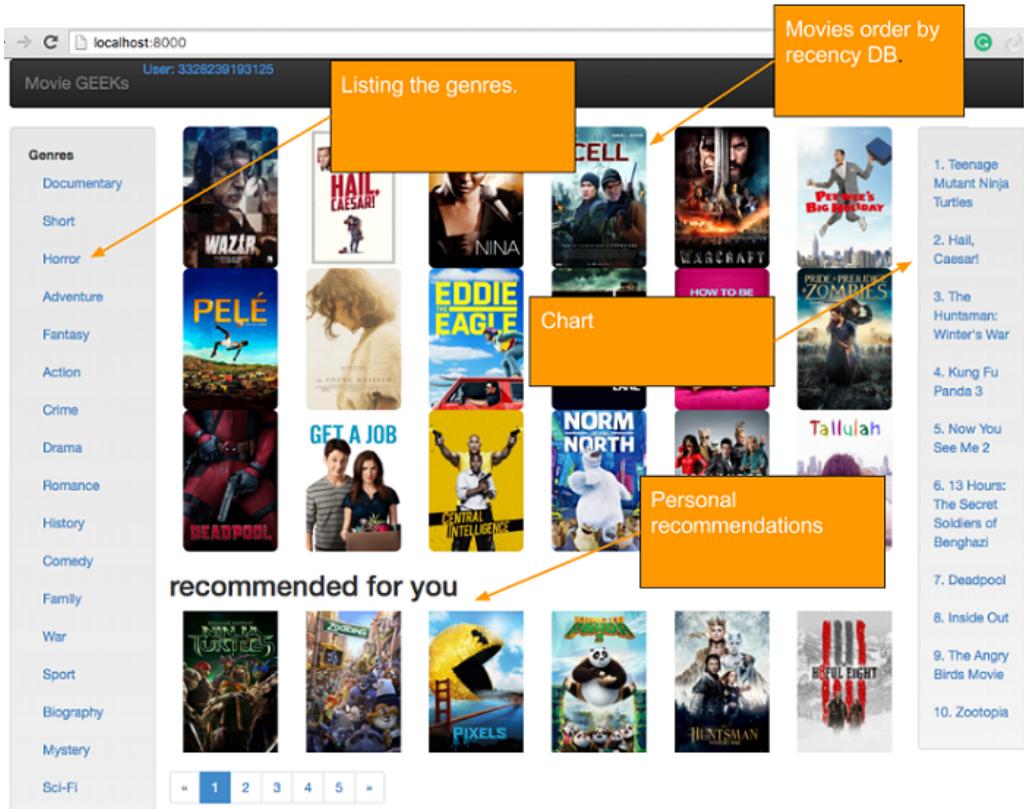


Figure 1.18 The loading page of the MovieGEEKs site

That's it! Simple but it will do the trick. Go ahead and download it now. Please refer to the readme on git hub for installation instructions.

<https://github.com/practical-recommender-systems/moviegeek>

The MovieGEEK site uses a dataset called MovieTweetings, it is a dataset consisting of ratings on movies that were contained in well-structured tweets on Twitter¹³.

1.5 Building a recommender system

Before moving on, let's have a quick look at how you would go about building a recommender system. Assuming you already have a platform in the shape of a website or some app where

¹³ <https://github.com/sidooms/MovieTweetings>

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/practical-recommender-systems>

Licensed to Asif Qamar <asif@asifqamar.com>

you want to add a recommender system, it would go something like the cycle shown in Figure 1.19

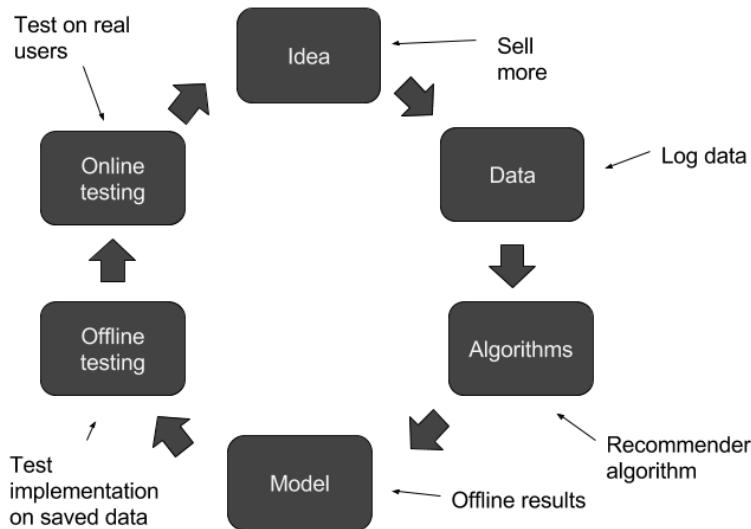


Figure 1.19 The data driven approach

Start out with an idea that you want to sell more by adding recommenders. You then (or hopefully you already have been) collect behavioral data, and use that data to build an algorithm, which creates some kind of model when it is run. The model can also be considered as a function, which will, given a user id, calculate recommendations.

With this model, you'd try it out on some historical data simply to see if you can use your model to predict some behavior of a user. For example, if you have data showing what users bought last month, then you can try to create the model using the three first weeks of data and see how well the model recommends things that the users bought in the last week of your month of data. If it is better at predicting what the users has bought compared to some baseline recommender system, which can be as simple as a method that returns the most popular items. If it is doing well, you can expose it to some of your users and see if you can track improvement. If yes, then it can go into production; otherwise, it's back to the idea.

1.6 Summary

The first chapter is finished, and you should now have an idea of what a recommender system is, and an understanding of what's needed as input and what can be produced. To summarize, you should now understand the following:

- Netflix uses recommendations to personalize its site, and help users select things they like. Recommender system is a common term for many different components and

methods. A prediction is different from a recommendation. A prediction is about predicting what rating a user would give some content, while a recommendation is a list of items that is relevant to the user.

- A recommendation context is what is happening around the user when a recommendation arrives. Items that might not be predicted to have the highest ratings can be recommended if they suit the context. The taxonomy described in the chapter is handy when you are looking at other recommender systems or trying to design your own. It is good to go through this taxonomy before starting to implement your own recommender system.

Knowing the basis of a recommender system gives you the foundation for chapter 2, which shows how to collect data from users.

Part 1

Getting ready for recommender systems

The environment is everything that isn't me.

- Albert Einstein

This part of the book will introduce you to the recommender system ecosystem and infrastructure. Using recommender systems, and in fact most machine learning methods in production, is not just about implementing the best algorithm, it is a lot about understanding your users and the domain. In this first part of the book you will learn how to collect data and how to use it when you add a recommender system to your application.

This part is about the environment around the recommender system algorithm.

2

User behavior and how to collect it

Welcome to chapter 2!

This chapter invites you to delve into the interesting subject of data collection:

- You'll start out by returning to the Netflix site to identify events, which could provide evidence to build a case for what a user likes.
- You'll learn how to build a collector to collect these events.
- With a collector, you'll learn how it is integrated into our MovieGEEKs website to collect events similar to the ones identified on the Netflix site.
- With a general overview in place and an implementation, you'll take a step back and analyze general consumer behavior.

Evidence is the data that reveals a user's tastes. When we talk about collecting evidence, we are collecting events and behavior that give an indication of the user's tastes.

Most books on recommender systems describe algorithms and ways of optimizing them. They start at a point where you already have a large dataset to feed your algorithms. You'll use one such dataset to create the example site MovieGEEKs. This dataset contains a catalog of movies, and ratings from real users. But a dataset won't magically appear-. Gathering the right evidence takes work and consideration. It will also make or break your system. There is a famous programming saying, "*Garbage in, garbage out,*" which is also true for recommenders.

Sadly, data that is good for one system might be unsuitable for another. Because of that, we will have a serious discussion about data that *could* be usable, but cannot guarantee that everything will work in exactly your environment. In this chapter, and throughout the book, we will look at many examples of how to approach data collection in your own site.

Generally two types of feedback are produced by users of a system: *explicit*, which are ratings or likes, and *implicit*, which is activity we record by monitoring the user. A user can provide explicit feedback in the form of a certain number of stars, hats, smileys, or some other icon illustrating how much that user likes a product. Usually the scale is between one and five (or one and ten). User ratings are usually the first thing people think about when talking about evidence. Later in the chapter you'll look at ratings, but they are far from the only thing that indicates what a user likes.

Ron Zacharski's *A Programmers Guide to Data Mining*¹⁴ presents a great example that illustrates the difference between ratings and evidence. He shows the explicit ratings of a guy named Jim. Jim states he is a vegan and enjoys French films, but in Jim's pocket is a rental receipt for Marvel's *The Avengers* and one for a 12-pack of Pabst Blue Ribbon beer. Which should you use to recommend things? I think it's an easy choice. What do you think Jim wants recommended when he opens the online ordering site for his local takeout place: vegan food or fast food? By collecting user behavior data, you'll be able to understand what Jim wants.

There's no substitute for good evidence. Let's have a look at what Netflix could record, and what they could interpret it.

2.1 How (I think) Netflix gathers evidence while you browse

Let's return to Netflix for an example of evidence. Everything on the Netflix landing page is personalized (the row headlines as well as their content), with the exception of the top row, which is an advertisement that everybody probably sees. The rows are different for each user; the headlines will range from familiar categories such as Comedies and Dramas to highly tailored slices such as Imaginative Time Travel Movies from the 1980s¹⁵. Figure 2.1 shows my personalized Netflix page. On my page, the first couple of rows contain recently added content, suggestions, and popular content. On that particular day, the first personalized row title is Dramas, which indicates that between the genres (or row headlines), Netflix thinks that drama best matches my interests. The Dramas row contains a list of content that Netflix considers interesting to me within that category. Figure 2.2 illustrates the Netflix content shown in my Dramas list.

Hovering the mouse over the row, I can make it scroll sideways, showing other content in the Dramas genre. If I scroll over the content of the Dramas row, what does this say about me? It could mean that I'm writing a book and trying to make screenshots, but most likely it suggests that I enjoy dramas and want to investigate the list further.

If I see a movie that looks interesting, I can place my mouse over it to view details. Moreover,

¹⁴ <http://guidetodatamining.com/>

¹⁵ This example comes from Netflix. I would love to know what's inside that category, as I'm sure I'd want to watch all the films, but besides the *Back to the Future* films, what could possibly be in there? Have a look at this article for more details on the colourful categories <http://www.telegraph.co.uk/on-demand/0/secret-netflix-codes-find-thousands-of-hidden-movies-and-tv-shows/>

if those details are intriguing, I might click to go to the page of the film. If it still sounds good, I'll either add it to My List or start watching it.

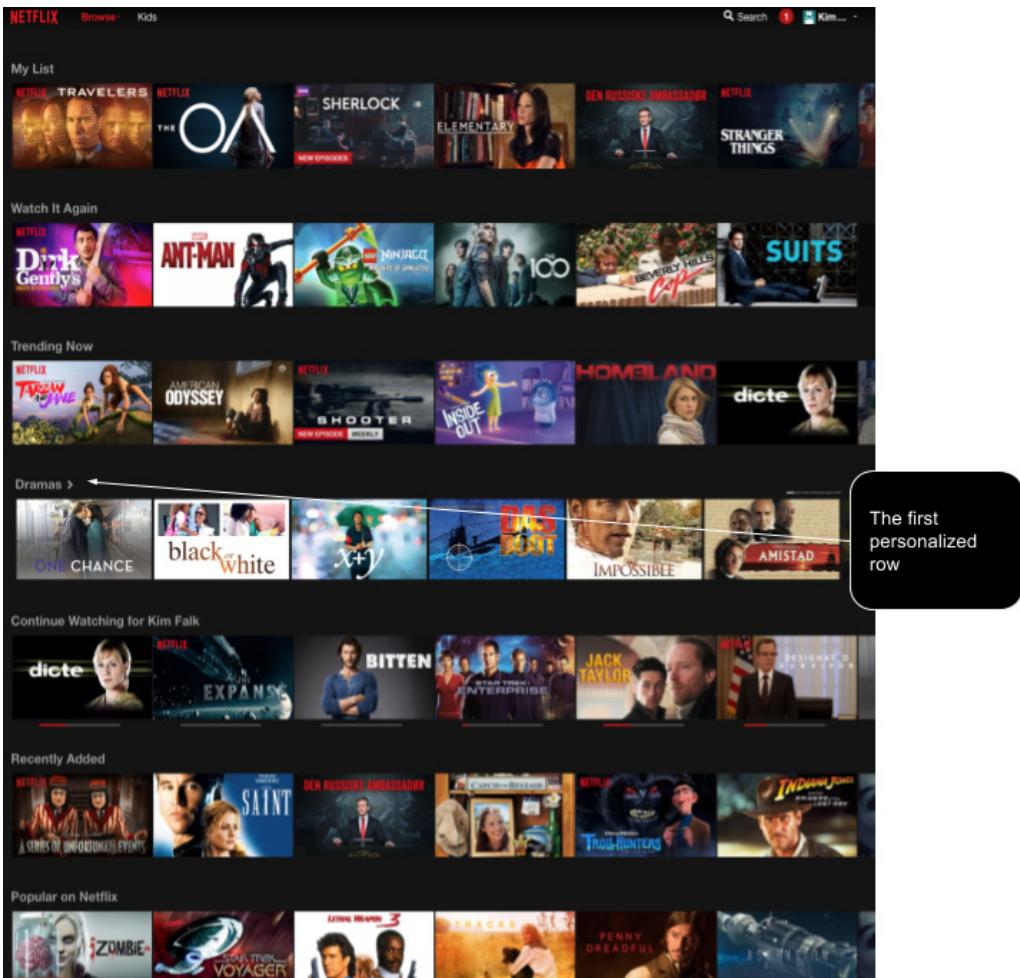


Figure 2.12 My personalized landing page on Netflix

You could say that for an e-commerce site, the act of watching (and finishing) a film is equivalent to buying something. But for a streaming site, this isn't where it ends. If a visitor starts watching a film, it's a positive event, but if he stops after three minutes and never restarts the film again, then it shows that the visitor didn't like what he saw. If he restarts the movie again later or within a certain timeframe, it might mean that he liked it even more than if he had watched it in the first go.

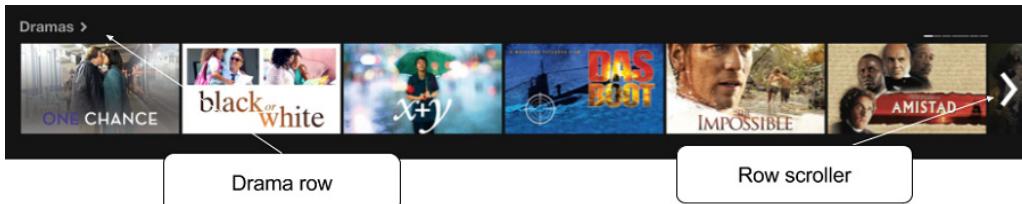


Figure 2.23 Netflix drama row, where you can either scroll the dramas using the arrow or click on the row title and get a complete drama listing

OFTEN THE PURPOSE IS NOT WHAT IT SEEKS

Often the purpose of e-commerce sites is to make people buy products, even if the product the customer is buying might not be exactly what he wants. That depends on the site's affiliation to the product. If Amazon sells you a bad T-shirt, for example, you'd say that the brand of the T-shirt is bad, but go back to Amazon and buy another T-shirt. If you buy a T-shirt from a Gap website, and it's bad quality, then you'll probably go to another site.

Subscription-based sites are a bit different. Mofibo, a Danish e-book streaming service, provides recommendations as inspiration and discovery, but with a catch: it is important to Mofibo that the reader knows what kind of book it is before starting to read it. Because Mofibo pays a fee every time a reader opens a book (not per page, but per book), so although Mofibo wants you to read as much as possible, it also wants to minimize the number of books you do it in.

2.1.1 The evidence Netflix collects

Let's try to imagine what goes on behind the curtains at Netflix and what data they collect.

Say it's Saturday night at Jimmie's place. Jimmie has user ID 1234, and after finishing nuking the popcorn in the microwave, he opens Netflix and does the following:

- Scrolls the Drama (ID 2) row
- Hovers the mouse over a movie (ID 41335) to get details
- Clicks to get more details about the movie (ID 41335)
- Starts watching the movie (ID 41335)

While he watches the movie, let's try to imagine what just happened on the Netflix server.

Table 2.1 shows some of the events that could be collected from this user, along with interpretations of what it could mean. Moreover, a column containing an event name is added to connect this table to the log described further down.

Table 2.1 Examples of evidence from Netflix

Event	Meaning	Event name
Scrolling a themed row	User is interested in the theme, here Dramas	genreView
Placing the mouse over a film (requesting overview of content)	User is interested in the movie (which is a drama and therefore also shows interest in this category)	details
Clicking the film (requesting details of content)	User is more interested in the movie	moreDetails
Adding the film to My List	User intends to watch the movie	addToList
Starting to watch the film	User "purchases" the movie	playStart

As the table illustrates, all these events are evidence to the system, because they uncover interests of the user. Table 2.2 shows how the evidence is probably recorded from a user who took the following steps.

Table 2.2 Example of how Netflix probably logs evidence

userId	contentId	Event	Date
1234	2	genreView	2017-06-07 20:01:00
1234	41335	details	2017-06-07 20:02:21
1234	41335	moreDetails	2017-06-07 20:02:30
1234	41335	addToList	2017-06-07 20:02:55
1234	41335	playStart	2017-06-07 20:03:01

There will probably be a long list of other columns such as device type, location, speed, and weather, which can all be used to better understand the user's context. I'd also venture that the number of log events for this scenario would be much more, but let's keep the example simple. The list of events types are probably much longer as well.

Now that you have a general idea about what evidence is, you can start looking into the implementation of an evidence collector. An *evidence collector* is used to collect data like that found in table 2.2. Just to ensure that you're not thinking that this can work only with media streaming sites, let's look at another scenario.

GARDEN TOOLS SITE EXAMPLE

I once had a colleague who would spend all his breaks surfing the internet for garden tractors or for anything that you could use in a garden and that ran on gasoline. Let's imagine this former colleague had a small break and opened up his favorite (imaginary) site called Super Power Garden Tools, with an angry logo featuring a lightning. He would do the following:

- Select the Garden Tractor category

- Click a green monster that can pull up trees
- Click Specifications to see the size of the trees it can pull up
- Buy the green monster

These events would be the same as the others; the importance of buying an expensive product might be more significant than starting to watch a movie, but I hope you get the picture.

2.2 Finding useful user behavior

Sites with high user involvement enable the site owner to collect large amounts of relevant data, whereas sites with mostly one-time visitors need to focus on relationships between the content instead. Don't despair if you don't have a streaming service with lots of user interaction to collect data from, chances are that there's still plenty to collect.

Ideally, a recommender system collects all data about a user when she interacts with the content—down to measuring brain activity, adrenalin released in the blood when exposed to an item, or how sweaty the user's hands get. The more we live our life connected, the more realistic this scenario might sound. In the movie *WALL-E*, humans have digressed into shapeless things that live all their lives in a chair in front of a screen, where everything about them is fed into a computer. (Come to think of it, I spend most of my days sitting in front of a screen. But at least I move between different screens.) But because most people also have other things to do besides being hooked up to a recommender system, we need to lower expectations a bit. With the web, we've become a lot closer to users than any physical store ever could, so it's possible to learn many things.

CONTENT AFFILIATION TO PROVIDER

In chapter 1, one of the dimensions in the taxonomy was the *purpose*. Purpose is important because it might result in particular strategies for calculating suggestions as well as what you want to suggest. Take, for example, a film; if you watch a bad film on Netflix, it shows you something about the quality of the content on Netflix and therefore says something bad about Netflix. If Amazon sells a Blu-ray disc of a bad film, you probably wouldn't think any less of Amazon; but if you were looking for the film but couldn't find it, that would make you think less of Amazon. I mentioned this already in section 2.1, but a good point is worth making twice. The purpose of Netflix is to show you good films that you like. Amazon shows you things to buy; whether you like them is not so important. Of course, Amazon spends many resources on making customers write reviews and rate content, so it might not be completely fair to say it doesn't care, but for the sake of example, it will do.

2.2.1 Capturing visitor impressions

To better illustrate the events that occur in the lifetime of a consumer/product relationship, I've divided it into the following steps, shown in figure 2.3. Here are the steps:

1. Consumer browses. Just as in a physical shop, the consumer looks around to see what's there, with no specific goal. What's interesting is where the consumer pauses and shows interest.
2. Consumer becomes interested in one or more products. It might be that the consumer knew from the start that she was looking for something specific or it might be simply by chance.
3. Consumer adds product to basket, or a list with the intention to buy.
4. Consumer buys products.
5. Consumer consumes product. The consumer watches the film; if it's a book, the consumer reads; if it's a trip, the consumer goes on the trip.
6. Consumer rates the product. Sometimes consumers return to the shop/site to rate the product.
7. Consumer resells product or otherwise disposes of it. The consumer lifetime of product is finished; it is disposed of, deleted, or resold, in which case the product will probably go through same cycle again.



Figure 2.3 Consumer/product relationship lifecycle

We will discuss what can be collected at each of these steps. But note that explicit feedback in the shape of a rating is done in step 6 or later, which is late in the process. Therefore, even if ratings is always the first thing people talk about, you ought to start recording data way before that.

2.2.2 What you can learn from a shop browser

Here, I'll go into detail about what's happening in steps 1–3 in figure 2.3.

A browser is a customer who browses through content. A browser might randomly go through lots of different things but will pause at content that seems relevant or interesting. In a physical shop, a browser strolls through, not showing any direction or purpose. In a sense, the customer is gathering intelligence for later buys.

A browser

A browser should be exposed to as many different things as possible, and suggestions should reflect that. If you could classify that a visitor is a browser, you could use that information to produce suggestions that fit that mood.

What you need to collect here is where the browser stops and investigates. It's also worth keeping track of what the consumer sees without showing any interest. But can you be sure that a page view (product view) is always good?

PAGE VIEW

A page view in a e-commerce site can mean many things. It can mean that the user is interested, but maybe somebody is just lost or clicking randomly. In those cases, more clicks weren't positive. They show a visit with lots of clicks, but no conversion in the end.

A great recommender would actually result in fewer page views, since people will find everything they need from recommended links and products without needing to browse around first.

PAGE DURATION

To figure out what a visitor browsing your site is interested in, you can measure the customer's duration on a content page. But is that straightforward? It is, mostly, if you assume that the customer isn't doing anything else, and that the next thing the customer does is to go to a new page by following a link on the current page.

Table 2.4 Page durations and a possible interpretation

Duration on page	What it means
Less than 5 seconds	No interest
More than 5 seconds	Interested
More than a minute	Very interested
More than 5 minutes	Probably went to get coffee
More than 10 minutes	Interrupted or went away from the page without following a link

Adjust the duration times to fit your domain, but I think most would agree that this could be true. Which of these are worth saving? Well, all of them. Less than 5 sec is a dislike, 5 sec to 1 min could mean “interested,” 1 min–5 min could mean the user thinks this is “great,” and 5 min and on is hard to say. All of these depend on the content of the page, and of course it’s not an exact science.

EXPANSION CLICKS

Besides duration on a page, there are other ways to record user interest in the content. Add small control interactions, which will help you determine what the user is doing. For example, websites often use links to more info, as shown in figure 2.4. This is convenient for the customer, who can get a quick overview or expand the link if interested. Similarly, a user might scroll down to see reviews or technical details, or download a brochure. If a user does one of these things, consider it a sign of interest.

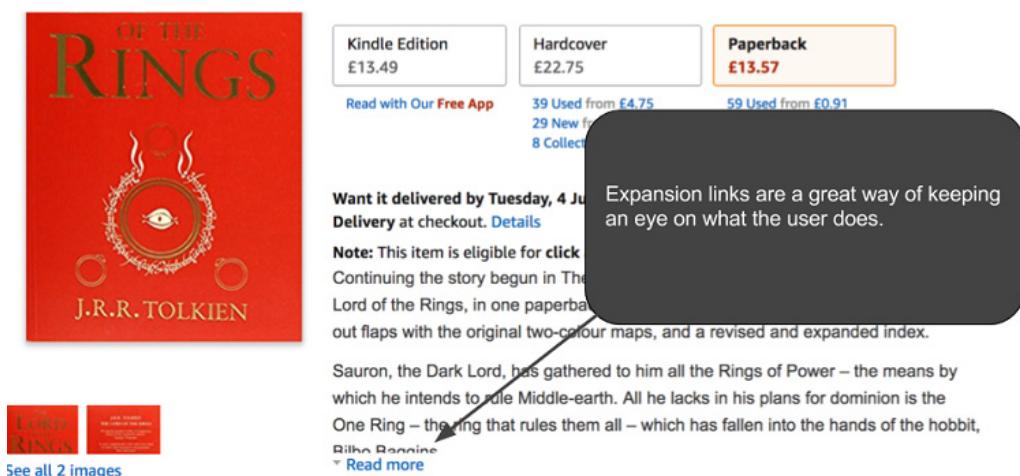


Figure 2.4 It is an indication that the user is interested when it clicks on an expansion link

SOCIAL MEDIA LINKS

With social media, there is also the possibility to add a Facebook-like button, [Share](#)     for people who like something so much they want to share it with other people. You can't control what happens on Facebook or Twitter or one of the other social media sites, but you can collect the event of a user sharing something.

SAVE FOR LATER

A *Save for Later* feature that enables users to add things to lists is powerful. If a customer finds something of interest, it's a good idea to provide the user the capability to save it for later (if they don't buy it immediately). This can be as simple as having a link for bookmarking the page. Even better, have a wish list, favorite list, or watch list, depending on the type of content.

Other signs of interest could be downloading a brochure, watching a video about particular content, or signing up for a newsletter on a specific topic.

SEARCH TERMS

Browsing a website can mean that people are either just browsing or that they're looking for something. If the page is good, most customers find what they want quickly. Netflix says that every time somebody starts searching it is seen as a failure of the recommender system, since it means people didn't find anything they wanted to watch right now between the recommendations. I am not sure I would agree on that, often I use the search function because something has been recommended to me, which might be outside of what I usually watch. In any case search is one of the best ways of understanding what a customer is looking for through their search terms.

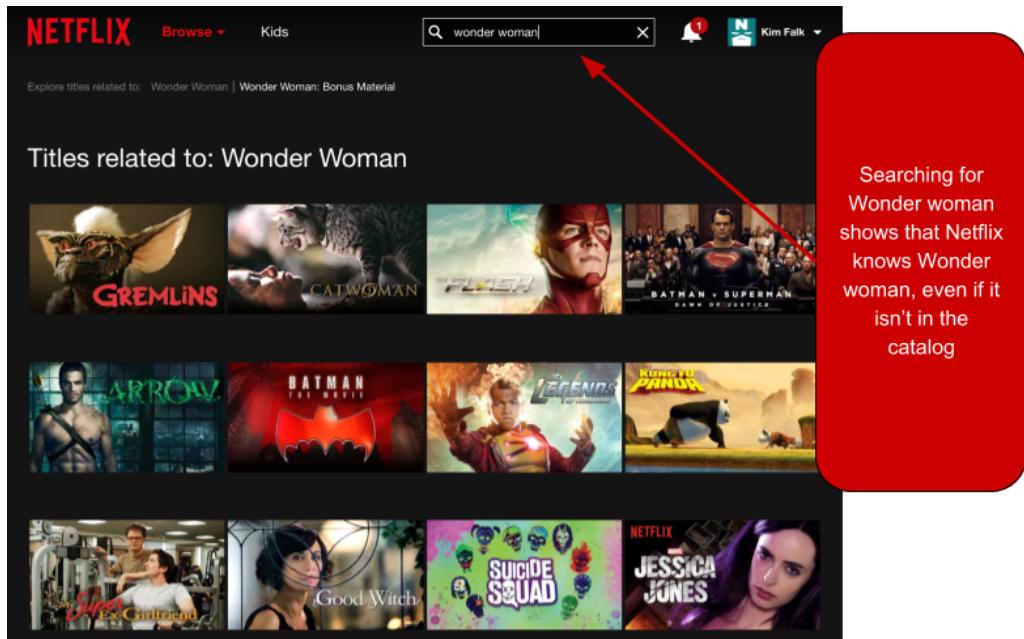


Figure 2.5 Netflix search result window, searching on *Wonder Woman*

Figure 2.5 shows a Netflix search window. The site has a lot more movies than it can show you. So if you search for *Wonder woman*, it will show you similar titles, even though *Wonder woman* isn't part of the catalog.

Even if the system can't provide the content searched for, registering the event is worthwhile. If a user looks for a film, you know they're interested in something about that content; so even if the site can't provide the film, it can suggest something similar.

Liking searched items with the resulting consumption

Another thing to consider about search terms (what a customer types in the search field) is that it's a good idea to connect what is searched for with what is consumed in the end. For example, say a user searches for *Star Wars* and looks at *Harlock: Space Pirate*, which has a reference to *Babylon A.D.*, and the user ends up watching that. Maybe it's worth putting *Babylon A.D.* in the search result for *Star Wars* next time somebody comes along searching.

2.2.3 Act of buying

Buying something means that the consumer considers the item useful or likeable – or maybe it is a present for somebody. It's not easy to figure out whether a purchase is for the buyer and thereby another piece of evidence which can be used to understand the user's taste, or if it is a present and something that should be disregarded.

Figuring out what purchases are presents and which are not is an interesting problem. An item that is different in taste from the items consumed by the user so far could either be an indication of a new dimension in the user's taste or a present for somebody. Either way it is what is seen as an outlier in the data. Graphically, an outlier will show up as points far away from the main body of points, as shown in figure 2.6.

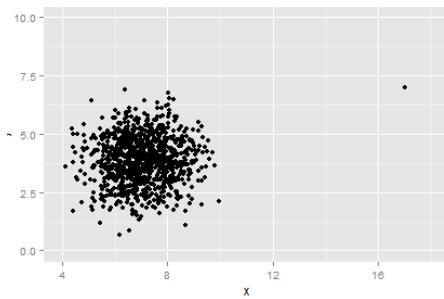


Figure 2.6 Example of an outlier

Since we can't be sure about what they indicated (present or a new interest), it is often better to disregard them. On the other hand, the outlier could also be a first indicator of a new trend, which is an opportunity that could be explored.

The act of buying something means that the product has been presented in a good way; it doesn't say anything about whether the consumer likes the product. At least, this is true if it's the first time the consumer buys the product. Some would argue that every time a consumer buys a banana, that indicates more stars on the rating for bananas. A first buy might not be much of an indication of approval, but a second buy is. But either way, a buy will be regarded as something positive.

2.2.4 Consuming products

When something is bought, the shop loses contact with the product and the ability to track how it's used—if it's not a streamed product or service provided from the website.

ENDOMONDO

Movies and music are not the only content that is consumed online. Endomondo.com is an example of a site providing online services; it offers a sport tracker. Endomondo.com provides a premium account for serious sports people. When you subscribe for the premium service, you will have a list of features, you can do the analogue with Netflix, at Netflix the features are the movies. Endomondo can keep track of how much you use the extra services, see whether you are using them. Collecting this data will help Endomondo understand their customers and enable them to recommend similar services, or understand where they should develop new ones. Telephone companies also have the ability to measure how consumers use their phone, they can track us in scary ways.

The following is a discussion on what you can learn from streamed products.

STREAMED PRODUCTS

In the case of streaming services, music, film, or even books, all user interactions can be considered an implicit rating. Listening to a song is an indication that the user likes it. But this

data can be analyzed even more. The following list shows user interactions in regards to music or movies:

- Start playing: The user is interested; that is already positive.
- Stop playing: Oh wait, maybe the user was curious enough to start playing but thought it was so bad that she stopped. Stopping a song within the first 20 seconds (or a film within the first 20 minutes) can be a bad sign, whereas stopping close to the end can be considered something else.
- Resume playing: Okay, forget about all the negative implicit ratings the system just registered; the consumer just had to go to the toilet or something. Resuming something after stopping it can mean various things. If it's resumed within 5 minutes, somebody or something probably interrupted the consumer, so the stop and resume should not be counted. But if the consumer stops and then resumes 24 hours later, the consumer probably likes the content a lot.
- Speeding: If the user skips something in the middle, it's probably not a good sign—but only if it's the first consumption. If it's the tenth time a film has been watched, skipping a boring scene probably doesn't make the overall perception of the film any worse. The technical proof reader of the book, noted that with music, he would often skip through songs to understand how a song is, if it is worth listening to. Songs have less context so you can get a sense of if you'd like the content, this wouldn't work in case of movies.
- Played it to the end: We have a winner; this is a good rating. The user sat through the whole thing. ("Played to the end" also means played until the film ends and the credits start rolling)
- Replayed: Replying content might mean something good for film and music, but for a site offering educational videos, it can also mean that the topic was too difficult.
- These steps work on most streaming products. How to collect evidence from streamed products depends a lot on the type of player that's used.

In the case of Endomondo, which is a streaming service, these steps and explanations don't really hold. In the sense, if you start Endomondo and indicate that you started running (by pressing the play button), it probably doesn't have anything to do with how much you like the app if you pause it after 10k, that just means that you should get in better shape.

Visitor ratingsFinally, you've arrived at what everybody talks about: the ratings.

WHAT IS A RATING

Netflix has a motto:

The more you rate, the better your suggestions.

Nevertheless, that might be a truth requiring modifications, as you'll see later. Most recommender systems use ratings, but those user ratings are usually weighted against user behavior. These systems use ratings only as a starting point. What you really want is to

capture the behavior of users. Many sites enable users to review content that they've viewed/bought/used. This enables the system to gain a better understanding of what users like and thereby what to suggest in the future. Often a rating is placed by adding a certain number of stars (or hats, smileys, or something else), but behind the scenes that's just a number on a scale. At Amazon, as in most places, they try to help you with what each number of stars means by providing tooltips. Figure 2.7 shows an example of a book review on Amazon.



Figure 2.7 When rating something on Amazon it provides hints as to what the number of stars means

As the user passes the mouse over the stars, a description is shown. In this case, four stars means "I like it." On top of rating something, some sites even enable users to write a review; TripAdvisor, for instance, uses reviews a lot.

Often you can say that a five-star rating plus a written review counts as more than a five-star rating alone, as the person who writes the review puts more thought into it. The same is true for one-star ratings. But if a person accompanies all ratings with a written review, then it doesn't mean more. Could buying something and not rating it indicate something about your preference?

A SENSE OF CONTROL

When just-add-water cake mixes first arrived in the stores, they were a huge failure. The product seemed perfect for busy consumers: the only thing they had to do was to add water.

But through consumer studies, the producers discovered that the problem wasn't that the process was easy, but that it was too easy. Baking a cake is about creating, but pre-preparing everything made the process too easy; it took away the consumers' sense of control. The producers said *all right, we'll let them add eggs also*. It's cheaper for us to produce, and the consumer feels empowered. When the cake mixes were introduced again, they were a huge success.



The reason that many sites enables users to add their preferences is exactly the same: to give the consumer a feeling of influence over what the system believes is that consumer's taste. However Netflix states that many people indicate that they like documentaries and foreign movies, but watch American sitcoms. What should Netflix suggest, then: things that make you feel bad about your choice of entertainment or things that you want to watch?

This is one of the reasons that it's hard to use datasets with ratings to test whether a recommender system is good. They can test whether your prediction calculations work, but not whether the system will attract more users.

SAVING A RATING

When a user adds a rating, it's a special event. But that event should still be saved among the evidence as any other event. It might also be worth saving directly in your content database, so you can show average ratings when you present the content to the user.

NEGATIVE USER RATING

Things get a bit tricky if you want to indicate that you hate particular content, because to review something you have to give it at least one star. Zero stars means no rating at all. If you hate something, you don't want to give it any stars, not even one. In a sense, not rating it is better than "I hate it", which you can indicate on amazon with one star, as shown in figure 2.8. Not liking something usually means not bothering to rate it. But if something really irritates you, you might want to release your frustration somewhere, and that's often in the form of a negative review.



Figure 2.8 At Amazon you show you hate something with one star. Not that I hate this one, if you want to get into Deep Learning, then this is the one to read¹⁶.

VOTING

Many sites have had success creating a community around users voting whether something is good. TripAdvisor is one, whose only service is the rating of hotels and restaurants. Another such example is Hacker news (<https://news.ycombinator.com/>), where the users are responsible for adding content, which can be links to articles and blog posts about “anything that good hackers would find interesting.” When something is added, you can up-vote it. The more up-votes it gets, the higher it will stand on the page (it’s *almost* as simple as that; you’ll take a better look at the algorithm later). Sites that use voting are called *reputation systems*.

2.2.5 Getting to know your customers the Netflix way

With a page that is almost completely built based on suggestions, it’s important to provide as much input as possible about your tastes. If Netflix thinks that your tastes are different, finding what you want to watch can be difficult. This is another point for people who claim that recommender systems are manipulative because the system does not provide equal opportunity to all content, and therefore it manipulates you. I understand the argument but disagree.

Netflix once offered the user the capability to assist in creating a taste profile¹⁷. The feature is not available longer. The Taste Profile menu, shown in figure 2.9, enables the user to Rate Shows and Movies; select genres by indicating how often the user feels like watching, for example, *Adrenaline Rush*, as shown in figure 2.9; or, finally, check whether the user’s ratings match his current opinions.

¹⁶ That is if you have a background in Machine Learning, otherwise you might be better off to start with Grokking Deep Learning (<https://www.manning.com/books/grokking-deep-learning>)

¹⁷ <https://help.netflix.com/en/node/10421>

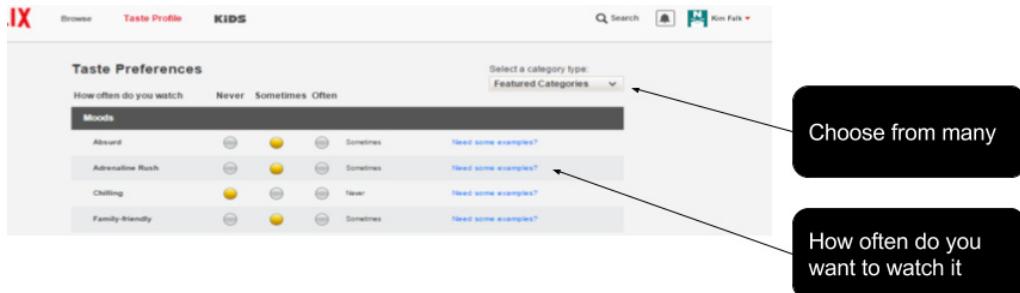


Figure 2.9: Netflix Taste preferences 2015, they have removed this feature now.

Netflix uses this manually input taste details to offer better suggestions. Asking the user for help with the taste profile is a method often used to enable the system to give suggestions for new users.

2.3 Identifying users

Collecting data on users works only if you have some way of uniquely identifying customers. The best way is to make the customer log in on your page so you have positive identification. Another alternative is to use cookies. Usually sites start out by setting a cookie, and connecting all info to that cookie. If the user then provides identification by logging in or by creating a profile, all info from the cookie is transferred to that account.

Be careful with cookies, as the computer could be a public or a family computer, used by several people. So saving data to the cookie ID over several sessions can be misleading. If you don't have logged-in users, there is also the crossdevice problem, which means that even if you recognize a user on one device, your system has little possibility to recognize the same user across different devices. There are services out there that will help you with this, but always with some uncertainty. Try always to make users register and login when possible. It comes without saying that personalized recommendations can only work if we recognize the user.

2.4 Getting visitor data from other sources

Your site is unique, and the data that can be collected on your site is the data that's best suited to reveal the behavior of your customers. But what if you could cheat a bit and get some data from somewhere else? Social media is a good place, and if you are lucky, a visitor on your site has liked something that matches the content in your catalog. Depending on what kind of content you have, the social media site could be Facebook, LinkedIn, or similar pages. Many will probably think that, yes, connecting to Facebook is good, but if you are not dealing with films and books, what's the purpose of adding data from Facebook? But most sites will benefit from getting something as simple as the age of a visitor or where they live.

If you keep an open mind, there might be other ways to gain knowledge. A recommender of pension plans could benefit from knowing whether a customer reads finance books, for example. That would be an indication that the customer is interested in the stock market, and will probably be more interested in pension plans enabling the customer to have more control over her portfolio.

Another thing to consider is that many algorithms calculate recommendations based on similar uses, so if the system has the same data about many users, even if that data might not be relevant for the content of the site, it will still enable the system to find similar users. I am sure a car dealership site can find that there's a special kind of film that most SUV owners like, or that a makeup site can recommend something based on age and gender. People who works in IT probably like gadgets instead of phones, and train drivers like sunglasses.

2.5 The collector

We are now going to look at an evidence collector implemented for the MovieGEEK website. We will look at the essential parts, and then leave it to the interested reader to go into detail with the code.

Because of performance and reliability of your website, it's better not to add evidence collection to your current (web) application. Instead, add it in a parallel structure that supports what you want to achieve. This enables you to move the evidence collector to another server if the users go wild after adding your recommender system and the load on your site gets close to the limit—or simply for scalability reasons.

This evidence collector has two logical parts:

- The server side: Which in our case is build using a Django web API that works as an endpoint that can be used by anything that a user is in contact with. Mostly that means a web page, but it could also be an app on a phone or any kind of device connected to the internet—anything that collects relevant events. When the server receives a notification that an event has occurred, its only job is to save it. A web API is an HTTP address configured to receive this kind of message, and can be implemented essentially by any type of framework.
- The client side: There won't be a client part in the traditional sense, as there are no web pages that can be requested. The client side consists of a simple JavaScript function, used to post evidence to the evidence collector on the server.

With a collector in place, you'll be prepared to start collecting data, either on our example site MovieGEEKs or on your own site.

To relate the evidence collector to the rest of MovieGEEK architecture, figure 2.10 highlights elements in the architectural diagram from chapter 1 with the evidence collector.

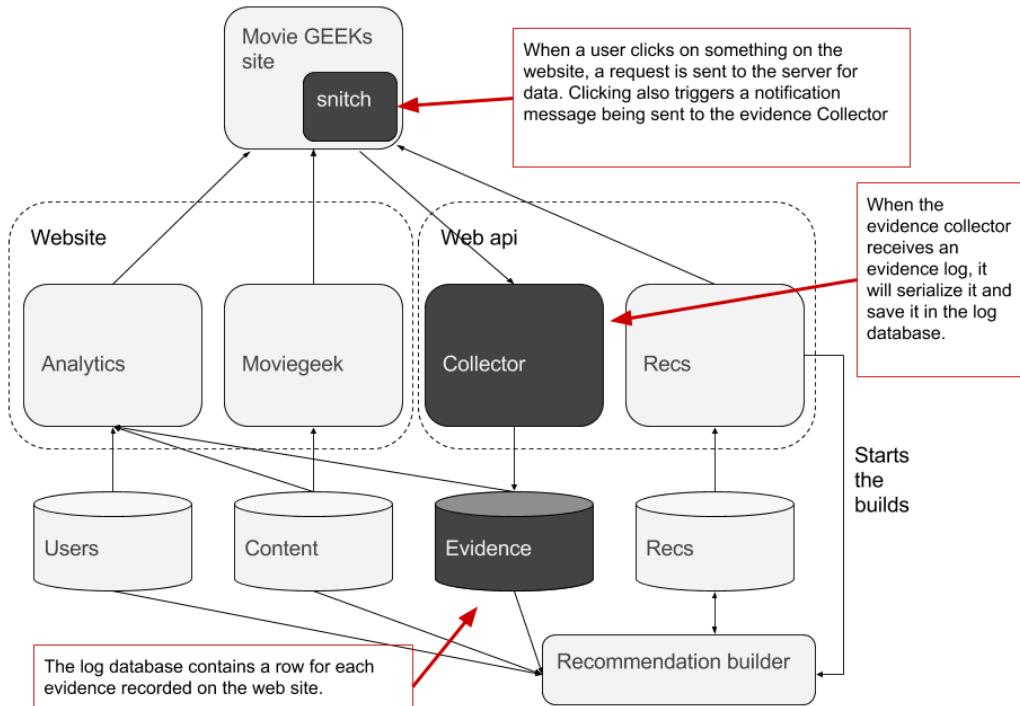


Figure 2.10 MovieGEEK's evidence collector and logger architecture

Some of you might be thinking, why not just use Django logging to save all the hassle of adding another app? But consider this:

- First, the logging in Django works on server-side code execution, so you'd have no way of saving user behavior (except by putting it in some kind of session state and saving it in the next request). All the events such as hover and scroll would be lost.
- Second, the collector enables you to receive evidence from everywhere, not just a website. The future is just around the corner, where evidence is something that you record from phone apps or other strange gadgets that are coming out, as well as from physical shops.

Because the data being collected is simple, it's often best to save it in a comma-separated file (CSV file). In this way, it's easier to move it around if the system that uses it is somewhere else. The csv file can then be fed to a service that ingests the data at the speed the system can handle. That way, you always have a buffer to ensure that you aren't overloading your system. But a CSV file isn't easy to query, and while learning about recommender systems, it's important to be able to query the data, so you'll instead use a database.

2.5.1 Build the project files

Getting the site up and running is fairly easy, you need to download or clone the following github repo:

<https://github.com/practical-recommender-systems/moviegeek.git>

After the download is finished, please follow the instructions in the readme.md file.

2.5.2 The Data Model

It is important to be able to collect most kinds of interactions. The previous section indicated that all you need are three things: user ID, content ID, and event type. You need a few more things to make it work, as listed in the data model shown in figure 2.11.

Log

date:	datetime
user_id:	char(64)
content_id:	char(16)
event:	char(200)
sessioninx	int

Figure 2.11 Data model of evidence logger

The session_id is used to uniquely identify each session.

Later, many things could be added to this, such as device type or context. But for now you'll start with this. There are some plumbing around the logger, which you are welcome to look into, but essentially it is just a web API, which is open for one type of request, containing the data above. We will virtually tag along on a request a little bit later on, when we talk about how it is hooked into the MovieGEEKs website. Let's have a look at what we should do client side.

A note on time

Recording time is always a bit tricky, because you have to take into account the different time zones. Using all local time zones means that you can have problems with the order of events, since events happening the same time in different time zones will be far apart. On the other hand, recording local times means that you can work with phrases such as "in the afternoon" on a global basis, rather than having to look at different time intervals for each time zone.

2.5.3 The snitch—client-side evidence collector

Evidence can be collected from anything that interacts with the user, from a phone app to a device you put in your shoe when going running.

The event logger is a simple JavaScript function, which simply calls the event collector. It doesn't do anything if there's an error, since it wouldn't be something end users could do anything about anyway. In a production environment, it might be worth keeping track of whether evidence is being recorded, but this is probably not the right place to do it.

The snitch should be in the project where you want to collect data, so you'll find file called collector.js in /moviegeek/static/js containing the following function, which creates an AJAX call to the collector.

Listing 2.1: /moviegeek/static/js/collector.js

```
function add_impression(user_id, event_type, content_id, session_id, csrf_token) {
    $.ajax(                                1
        type: 'POST',                      2
        url: '/collect/log/',             2
        data: {                           3
            "csrfmiddlewaretoken": csrf_token, 3
            "event_type": event_type,       4
            "user_id": user_id,           4
            "content_id": content_id,     4
            "session_id": session_id     5
        },
        fail: function(){                6
            console.log('log failed(' + event_type + ')')
        }
});};
```

- 1 Do an ajax call.
- 2 To be RESTful the message sent is a POST since we are adding something to the db.
- 3 CSRF middleware token is used to allow our site to call a site from a different domain.
- 4 The three important data elements.
- 5 And a unique session ID
- 6 If it fails, then write something out to the browsers debug console. Don't show the user anything.

The calls from the function in code listing 2.1 will eventually be received by the log method shown here

Listing 2.2: /moviegeek/collector/view.py

```
@ensure_csrf_cookie
def log(request):

    if request.method == 'POST':          1
        date = request.GET.get('date', datetime.datetime.now())          2

        user_id = request.POST['user_id']
        content_id = request.POST['content_id']
        event = request.POST['event_type']
        session_id = request.POST['session_id']

        l = Log(                                3
            created=date,
            user_id=user_id,
```

```

        content_id=str(content_id),
        event=event,
        session_id=str(session_id))
    l.save()
else:
    HttpResponse('log only works with POST')
return HttpResponse('ok')

```

- ① this method is only interested in POST type messages.
- ② create a timestamp to add in the created field.
- ③ save a log entry in the db.
- ④ respond nicely even if it is not a POST message.

④

2.6 Integrate the collector into MovieGEEK

The MovieGEEKs app cover the examples shown in table 2.5. The examples are similar to the ones listed in table 2.1, which depicted a use case on the Netflix site. I hate skipping back and forth, so I've added the table again here, with a new column showing the event data that will be collected.

Table 2.5 Movie GEEKs evidence points

Event	Meaning	Evidence
Clicking a genre, such as Drama	User is interested in the theme (here, Dramas)	(Kimfalk, drama, genreView)
Placing the mouse over a film such as Toy Story (requesting an overview of the content)	User is interested in the movie	(Kimfalk, ToyStory, details)
Clicking the film (requesting details of content)	User is further interested in the movie	(Kimfalk, ToyStory, moreDetails)
Clicking Save for Later	User intends to watch the movie	(kimfalk, ToyStory, addToList)
Clicking the Buy link.	User watches the movie	(Kimfalk,ToyStory, playStart)

LOGGING GENRE EVENTS

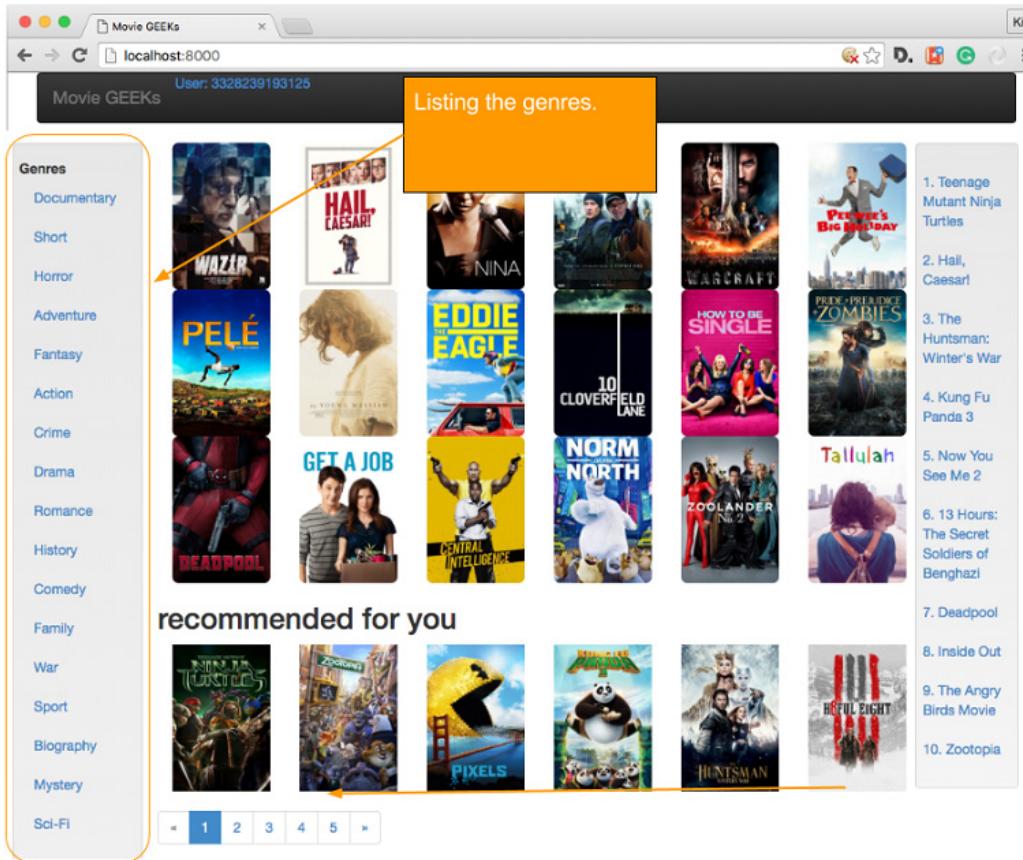


Figure 2.11. MovieGeek front page

The following events have been implemented in the `templates/moviegeek/base.html`. The first event to log is the user clicking a genre. The genres are listed on the left side of the screen, as shown in figure 2.12.

To register clicks on a genre a `onclick` attribute is added to each link. When the active user clicks on a genre it will fire a HTTP post request to the collector. We wont go too much into the code as it is quite repetitive. But have a look at the following figure to understand what happens when a user clicks on a genre.

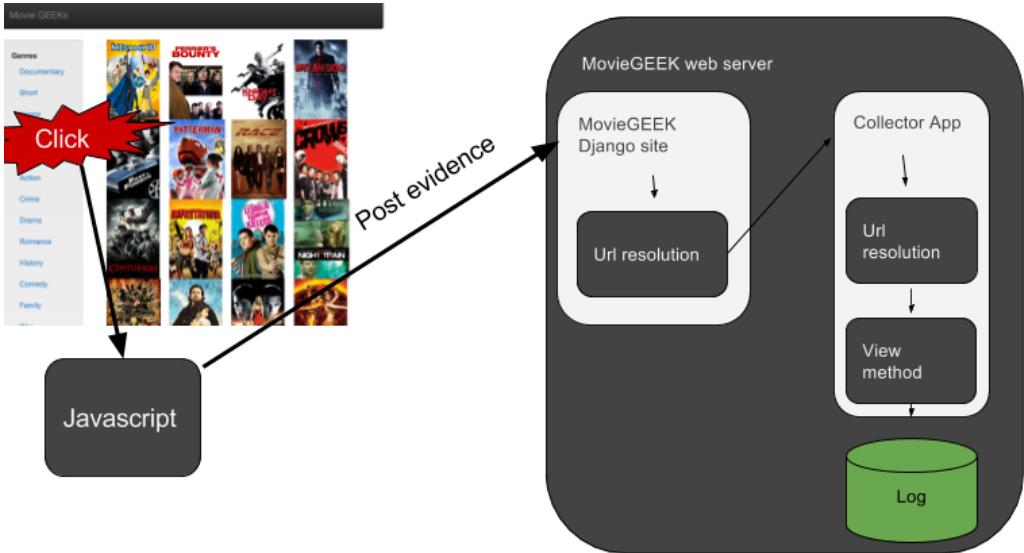


Figure 2.12: What happens when a user clicks on a genre

- User clicks on a genre
- The onclick event is executed, which calls the javascript function in Listing 2.1.
- The add_impression function is executed.
- HTTP request is received by the web server which delegates it to the moviegeek site.
- Moviegeek site does a lookup in the url list, and everything that has the url "/collector/" should be delegated to the collector app.
- The collector app matches the "log/" to a view method
- The view method creates a log object.
- Log object is saved to the database, using the Django ORM system¹⁸.

LOGGING POPOVER EVENTS

When a user clicks a film, a popover appears. A popover is a fancy name for a big tooltip, figure 2.14 shows how this looks. The fact that the user clicks, indicates that the user might be interested in the film, and is therefore something you should log. This is done by adding an event handler which calls the collector every time a popover is shown.

¹⁸ <https://docs.djangoproject.com/en/1.9/topics/db/>

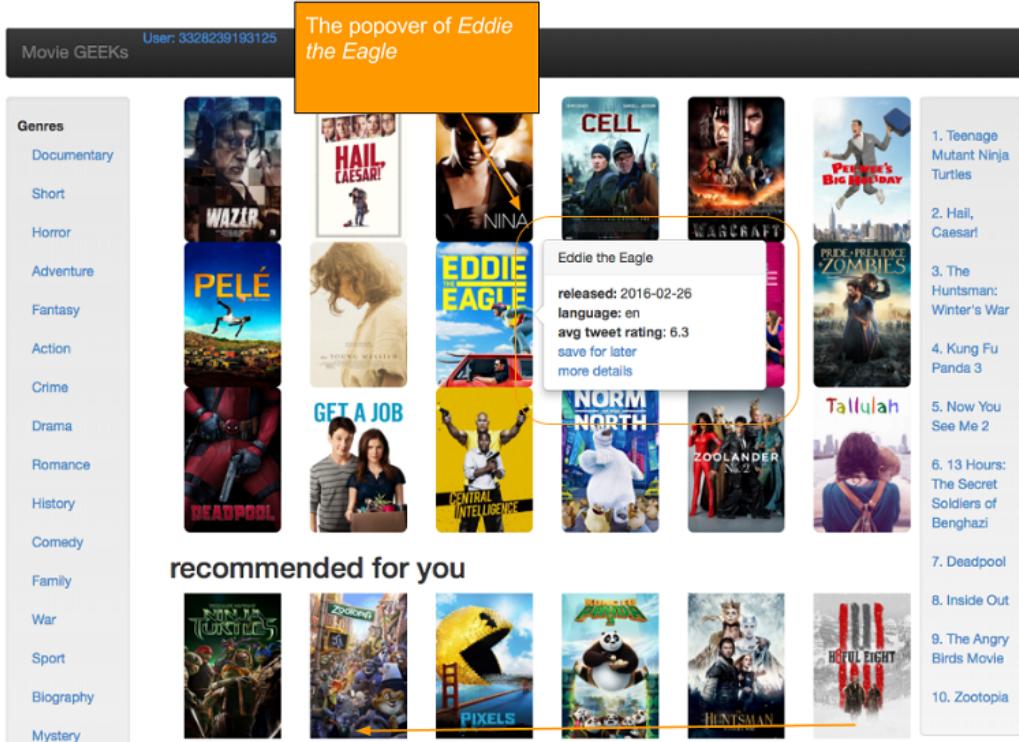


Figure 2.14 MovieGeek front page with popover

LOGGING MORE DETAILS EVENTS

A user who finds the information in the popover box interesting can click the More Details link. This is also interesting for you, as it shows further interest in the movie.

LOGGING SAVE FOR LATER

Instead of clicking on more details, the customer can also add something to a list, that's an important event, as it indicates that the user is planning to buy/consume it later. So the functionality is good to have; it's a link on the details view, which will record an event, which we will call `saveforlater`.

There are other things we could record also, but these will suffice for our purpose.

2.7 What is a user in the system And how to model them

Before moving on we need to think a bit about users also. We have talked about behaviour of users but beyond that what other things might be useful to think about how we will represent

other knowledge of users. Because as mentioned earlier, many other things can be relevant to know about the users. In other words, we need a user model that we can translate to a database table and use in addition to the evidence.

So what could be relevant to know about a user? Again, the answer is as, always, it depends....

I hate people answering "it depends" since it is one of the most useless replies, up there with "yes, but no", "it was once true, but not now". So assuming you are of the same conviction let's pretend we didn't just answer that and look at different scenarios where we can say something about what would be relevant. Another answer could be "everything" - in theory, everything could be pertinent to recommend things.

If we are implementing a recommender system on a JobSite, then it would be relevant things like current position, education, years of experience etc. If we, on the other hand, are looking at a pension site, we probably need the same things as the jobsite, but then health data such as how often you have been to the hospital, what medicine you are taking will also be of interest. Book sites could also use all the things mentioned above, as these are all things that will say something about book you'd be interested in reading. But most probably book sites will use just things like taste, buying habits.

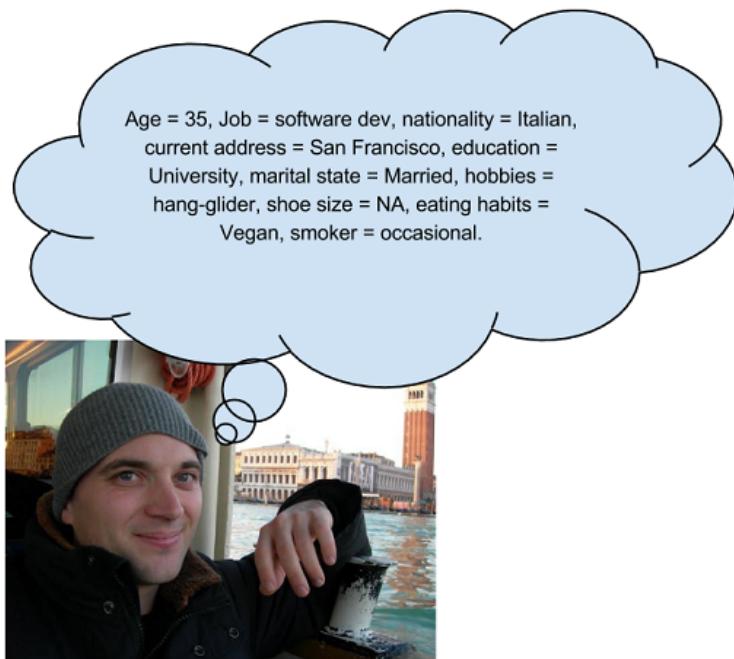


Figure 2.15 A new user called Pietro

So let's say "everything" and take it from there. Let's say that Pietro is being created in our system (see Figure 2.15), what information would be good to store on Pietro?. Given we had the possibility to retrieve the following information, what should we save in our database. In this day and age where storage is so cheap so why not save all of it, and let's see what we can do with it in the future chapters.

In other words, ideally we would want to keep a list of key-value pairs, next to the identifiers for the user, like the user id, email address possible some other things. Again remember we are doing a recommender system, typically you would also save shipping address, and things like that, but for us, for our purpose this is not so important. That would give us a data model like the one shown in Figure 2.15

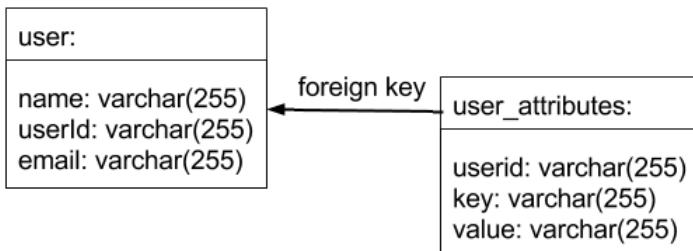


Figure 2.15: Generic data model

However, we want several things here. We want:

- flexibility to save everything, and
- Simplicity for making code readable

We will, therefore, do a less flexible, but easier to use implementation somewhere in the middle ground of what we said above. We will create a table containing most of the attributes above, plus one that contain any extras we might need later. So our data model for a user will look like Figure 2.16.

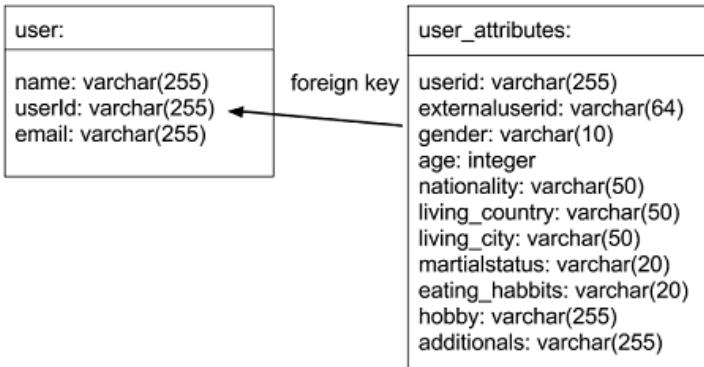


Figure 2.16: User data model

Continuing with the Pietro, we will save the data shown in Figure 2.17

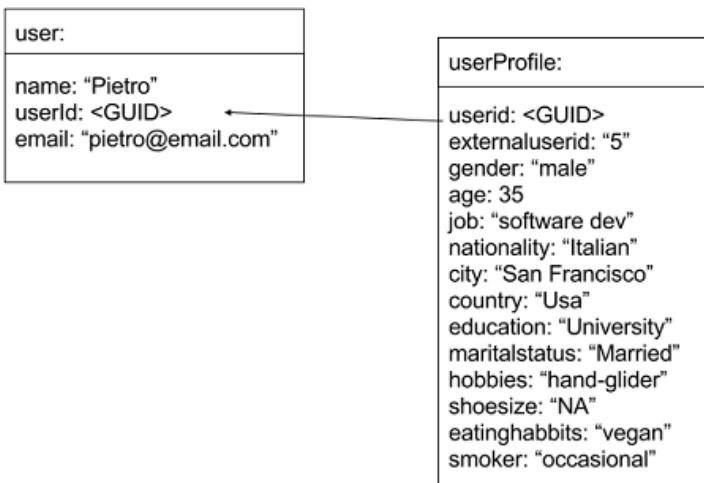


Figure 2.17 : Data collected for user Pietro

This data model is not very flexible, but on the other hand, until we need a flexible data model, its better to keep the level of complexity as low as possible.

2.8 Summary

In this chapter, you have learned about how users interact with items. You should now be able to explain to your rubber duck the following things:

- Log user behavior, using a web API, possibly running on a different web app than the

site, to ensure that it won't cause the site to suffer in performance if the user triggers many events.

- Connect a snitch to a website by attaching a call to all events happening on a site.
- Good evidence are things that provide information to the system about users' taste. It is good to record all events, they might turn out useful later.
- Implicit ratings are what is deduced from the events triggered by the user, while explicit ratings are the actual ratings a user inserts.
- Implicit ratings are usually more reliable, but only if you understand what each event means.
- Explicit ratings aren't always too reliable, because they can be biased due to social influences.

If you have regular visitors, your log might already have something to show you about your visitors. Nevertheless, it's always good to have some stereotypes to check how your algorithms work. The following chapter introduces personas and shows how to autogenerate evidence.

3

Monitoring the system

This is one of the shorter chapters but still contains a lot of knowledge:

- You will begin with analytics, all data driven applications should start out with analytics.
- I will attempt to convince you of the great value of analytics, and we will look at how to implement an analytics Dashboards.
- You will have an introduction to Personas, and why they are useful.
- Using these Personas, you will go through some different ways to represent user taste.

So far, you've looked at what a recommender system produces and what you can learn from users visiting a site. At this point, you should understand what you want to achieve, and what evidence you need to build upon. Now you're missing only the part in the middle, shown in figure 3.1.



Figure 3.1 Data flow from evidence to recommendations. You start with your evidence, which you can aggregate into website usage. With that, you can start understanding the user's taste, which finally can work as input to the recommender system to produce recommendations.

To understand the two middle steps, you need to find a way to understand what your users are up to. You need to find a way to reduce the data for each user to a preference.

To do this, we will auto generate interactions, which will provide us with data we can base our discussion on. You're going to do some simple analytics. I've always thought of analytics as a continuous poor-man's data analysis; it does give you some information about your data, but it only scratches the surface of what's happening. Nevertheless, in our case that might be enough, as you'll see in this chapter.

The reason for resorting to analytics in a book on recommender systems is that you need a way to understand what's happening and to measure whether all your efforts will have any impact. Of course, making recommender systems is such fun that you might not care. But the people who are paying you will want to know.

You'll begin your journey into analytics by learning a little bit of the theory behind what you want to show. Then when you've seen the light, we will look at it in the context of MovieGEEKs site.

3.1 Why adding a dashboard is a good Idea

Back when I earned my degree as a computer scientist, we used to make fun of the students working with visualizations, calling it "circus computer scientists."¹⁹ So after University, I was never a big fan of anything containing colorful graphical interfaces. I preferred the undisturbed "beauty" of data in a table.

But as I got into bigger datasets and data analytics, I saw the errors of my ways and understood the importance of data visualization. After working with large data for many years, I've become a firm believer that running a data application without any visual representation is like driving with your eyes closed: you feel great, but you will realize that something is wrong only after it's already too late.

3.1.1 Answering "How are we doing?"

Let's start out with a question: "How is your website doing?" How would you answer that? Where's your focus? Is it money? Or number of visitors? Maybe average response time? Or something different altogether?

By adding recommender systems, what do you expect to improve? The first result will be that your colleagues will be much happier, because working with recommender systems is the best thing since the invention of web shops. But what will it actually change? That's the question of this chapter. Having a data dashboard will be paramount in understanding how your website is doing, how you can use the data you are collecting and if the site is improving.

You want to start with a benchmark showing the current performance. You'll leave some space on the dashboard to allow it to evolve with other markers as you progress.

¹⁹ In Danish, it's "cirkus datalog," which sounds a bit better than circus computer scientist.

This is so important that I think that you can't implement a recommender system without having an analytics dashboard to keep an eye on things. I therefore recommend that you build the analytics part of your site before starting to add a recommender system.

In the following, we will look at a dashboard implemented with MovieGEEKs, showing an example of how you can track the stream of events and customer behavior. First, let's talk a bit about what you want to achieve.

DASHBOARDS

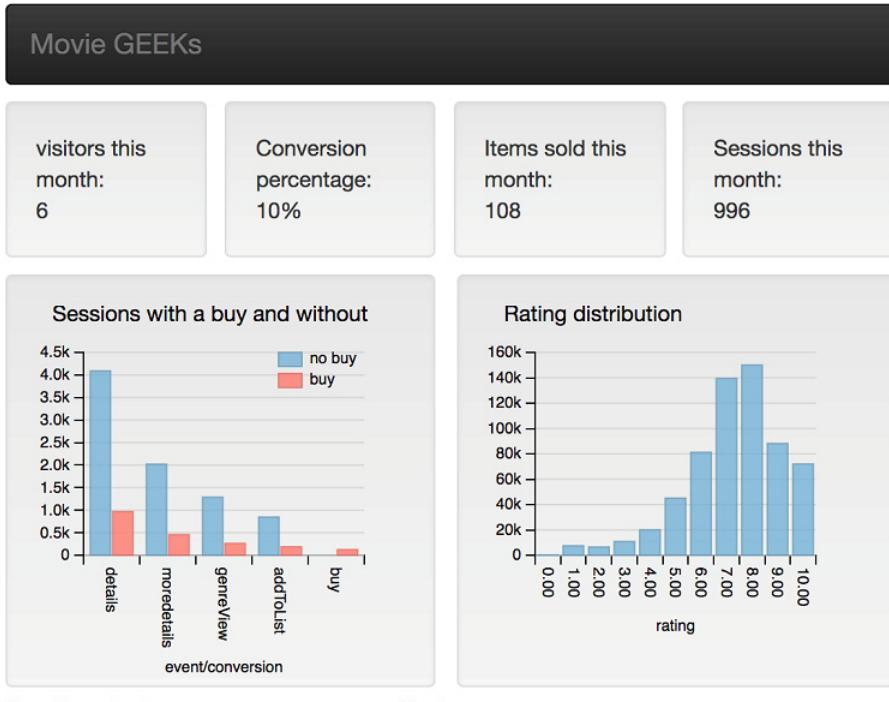
Most companies aren't too happy about telling the world how they track behavior or monitor performance. There are many reasons for this; two good ones are that it can be a business disadvantage, and it can give hackers hints as to what weaknesses they could have. In addition, if your users know too many intimate details of your recommender system algorithm, the user's behavior could become less spontaneous. This could induce biases in the results or even make users do things to push certain recommendations in a specific direction.

But for you, understanding your users is paramount. In this time when everything should always be more personalized, site owners need to know their users and what they're doing on the site, to enable the people to do data driven decisions and react quickly if anything changes. Visualizations will help you gain a better understanding of the data.

Mind you, what you'll be doing in this chapter is only the beginning, and should be extended when you get a better understanding of the system!

OUR DASHBOARD

Our analytics dashboard will look like the one in figure 3.2. You'll look at how to measure the performance of the recommender. That will add more to the dashboard.



Top 10 content

Clusters

- Teenage Mutant Ninja Turtles: Out of the Shadows (5)
 - Be Somebody (4)
 - Now You See Me 2 (4)
 - La La Land (4)
 - Jackie (3)
 - Logan (3)
 - The Secret Life of Pets (3)
 - Kung Fu Panda 3 (3)
 - Snowden (3)
 - Office Christmas Party (3)

Figure 3.2 MovieGEEKs analytics dashboard

3.2 Doing the analytics

Whether a website is performing in measures of response time or responsiveness is a significant factor in the success of the site. But, this is the topic of so many books. Here we'll

be concerning ourselves with the business part of a website, as that will tell you whether you're showing good recommendations or not.

Disclaimer

No matter how good the recommender is, it will only be as good as the content. No matter how good the recommender is, it will never be able to recommend anything to a vegetarian at a butcher shop.

3.2.1 Web analytics

What you're going to do is often called *web analytics*. Web analytics is split into two categories: off-site and on-site. *Off-site analytics*, which is about the potential of the website, focuses on opportunity, visibility, and voice. *Opportunity* indicates how big the potential of the site is, in terms of the number of visitors the sector has in total; *visibility* indicates how easy it is to find the site; and, finally, the *voice* indicates how much people are talking about the site.

This chapter concentrates on *on-site analytics*, which is concerned with how visitors behave on your site. In this category, the focus is on conversions, drivers, and Key Performance Indicators (KPIs), which are explained in the following section.

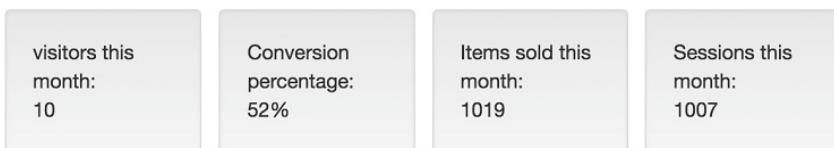


Figure 3.3 The KPIs in the top of the analytics app

3.2.2 The basic statistics

Analyzing the evidence you've collected might not give you an Indiana Jones magical sense of adventure, especially if the data you're looking at is generated data, as described in the previous section. In this case, you want to implement a visualization of the evidence, which will enable you to analyze the data collected. Often, you'll find this under the name of *summary statistics*.

The top row of the dashboard shows important numbers about the current state of your site, the KPIs. KPIs can basically be anything that evaluates the success of your website. So, what might that be? Firstly, it's important that people come and visit (because on the internet, it's not "Build it and they will come"), so the first KPI is the number of visitors. Next, you have the conversion rate, which is described in more detail later in this section. Then you have the number of sold content this month. Possibly numbers regarding money, such as total revenue, are also important, but that's not important for us right now.

The numbers are configured to provide statistics for the last month, but they could be daily or weekly, or even hourly if you have enough traffic, and time to sit and watch. You can also use sliding window, displaying statistics from (now - 1 month) until (now). Alternatively, you

can say this month, this week, and so forth. It's not terribly important what you do, as long as it's consistent.

3.2.3 Conversions

Imagine that you have a website for a new religion, and you could subscribe to the religion by paying a monthly subscription. When a person signs up, that's a conversion, in several senses. Surprisingly, the type of conversion that you're interested in is that the user is converted into a paying customer. In marketing lingo, that's called a *conversion*.

In electronic commerce, conversion marketing is the act of converting site visitors into paying customers.

I know you're all here with pure intentions and ambitions to create the best customer experiences you could ever dream of, and the mere mention of KPIs and business conversions are words of disgrace that make you want to slam this book shut—but wait a minute! You're back at the *purpose* again, which is what marketers try to measure with conversions and KPIs.

Conversion rate is an often-used KPI that's defined as follows:

$$\text{Conversion rate} = \frac{\text{Number of goal achievements}}{\text{Number of visits}}$$

The conversion rate is a dear child and has many names. You'll find it called the *goal conversion rate*, when we talk about goal completions, or *commerce conversion rate*, when a goal is a transaction in which something is bought.

Want to make your colleague the marketer happy? Say that you can calculate the website's conversion rate. (Just look for the colleague who looks like Stef: figure 3.4.) Online marketing is all about conversions. When a user does what you want him to do, you say that he converted or that you had a website conversion. Conversions are defined differently by individual marketers and content creators, and can be many things, usually related to selling something. For content creators, a conversion can also be about indicators that their content has been read—such as users signing up for a subscription, a newsletter, or a software download, or filling out a lead/contact form. But be careful about sounding too confident, because next your colleague will start talking about ROI (return on investment), and that isn't covered here.

To make things even more tricky, the word comes from the idea of a conversion funnel. A *conversion funnel* shows the path users take before they become customers. Let's try to imagine how a funnel might look like for a company such as Amazon. Figure 3.5 illustrates a funnel. A lot of people open Amazon's home page. They have a look, and might click around a bit. Some will move on to look in one of the departments, such as clothing, and might search for shirts and go through the results before finally selecting a funky Hawaiian shirt.



Figure 3.4 Stef the marketer

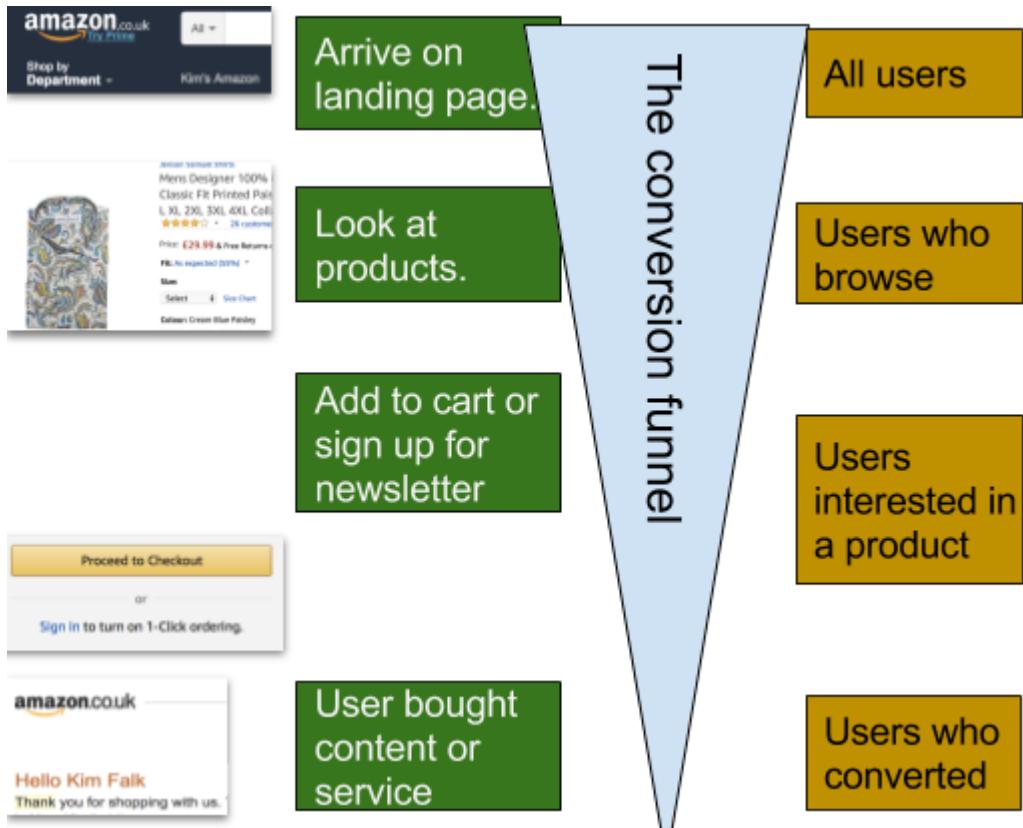


Figure 3.5 Conversion funnel

Each event that pushes the user a step down the funnel is called a *value event*. A value event can be many things. At Netflix, it can be that a user watched a film. On Match.com, a wink is a value event on your journey to eternal love or paying your next monthly subscription.

It's called a *funnel* because it starts out wide and then narrows down. The same occurs with site visitors: a (hopefully) huge number of visitors come to your site; some of them will add something to a wish list, some will share a product, and finally some of those will buy something.

Converting is marketer lingo, but what if we turn it around and look at it from a user's point of view? A conversion is also what the user wants; a user who comes to Amazon comes to find something to buy, or to sign up for an interesting newsletter. You're free to claim what you want, even that you're being manipulated or pushed, but the fact of the matter is that Amazon is a place for buying content, so if you go there, it's mostly for buying, and as such, you should be helped to get to the best possible things to buy. Later you'll look into topics that

don't align with the goals of these two groups, but for now let's stay in the world where everything comes together and people are holding hands.

In our system, the evidence collector is capable of registering conversion events. You can now have a quick look at how your system is doing, simply by asking the conversion rate. Now, the goal can change a lot; on a car dealership site, you wouldn't expect a customer to convert on every visit, whereas on Amazon you'd want most visits to end in a conversion (and it does for me, at least). Think about which of the two images in figure 3.7 depict a match to your site.

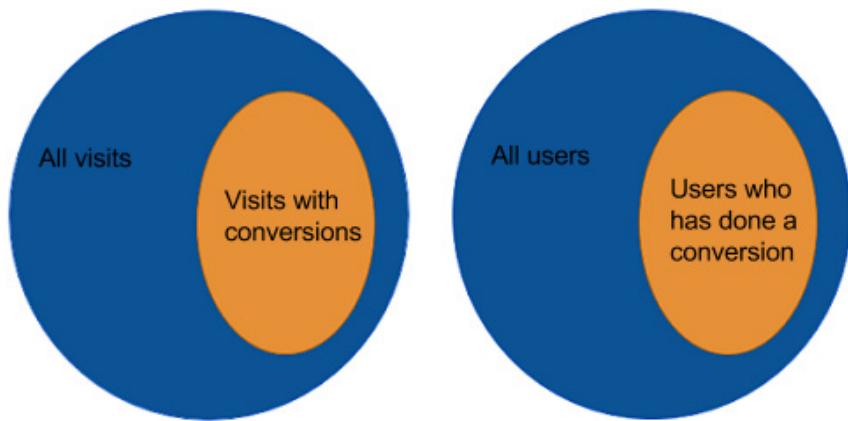


Figure 3.6 Different ways of looking at conversion

If you chose the left, it means that you have an Amazon type of website. Your hope is that a customer makes a conversion per visit—or at least somewhere close. If you chose the right one, you're more like the car-dealership site; you want to have a visitor converting in the lifetime of her contact with the site. For you, it's not important whether a customer visits the site hundreds of times over a period of time, only that they buy a car in the end. In this second case, the conversion rate is calculated using this formula:

MovieGEEKs is a site of the first type. Looking at the set of visits, the conversion rate will be the number of sessions in which a buy event occurs, divided by the total number of session IDs. This translates to the following SQL.

Listing 3.1: SQL script to calculate conversion rate

```
select count(distinct(session_id)) as visits,
       count(case when conversion > 0
                  then 1 end) as conversions
  from
( select session_id,
          sum(case when event = 'buy'
                     then 1
                     else 0 end) as conversion
    from collector_log
)
```

```
group by session_id
) c
```

- ① Counts the cases where a session contains one or more conversions
- ② Inner query, to calculate for each session whether a conversion had happened in it

Using the latest generated data, this returns the following: “209;116”, which means that the conversion rate is more than 50% (more exactly, 0.56). As mentioned earlier, it might be difficult to relate to this number, but keep it in mind, because this is the rate that you want to improve with your recommendations.

3.2.4 Analyzing the path up to conversion

You’re happy when a conversion happens, and the purpose of the recommender is to make the user convert. Now if you do a query to find all the event types stored in the database, along with a count of the number of times those events occurred, the results could look like the following:

genreView	1527
details	4953
moredetails	2034
addToList	976
bBuy	510

The following listing shows the query.

Listing 3.2: SQL calculating the distribution of events

```
select event, count(1)
from collector_log
group by event
```

This gives you a hint as to how often each type of event occurs. What you need to look at again is how often these events happen in buy sessions and in non-buy sessions. This can be checked using the following query, which groups the sessions such that for each session a row indicates the number of `buy`, `details`, and `moredetails` events that happened for the session.

Listing 3.3: SQL script that calculates how many times each event happened in each session

```
select session_id, 1
      sum(case when event like 'buy'
                then 1 else 0 end) as buy, 2
      sum(case when event like 'details'
                then 1 else 0 end) as details, 3
      sum(case when event like 'moredetails'
                then 1 else 0 end) as moredetails 4
   from collector_log
  group by session_id 5
```

- 1 Get the sessionId
- 2 sum all the buy events in the specific session
- 3 sum all the details events in the specific session
- 4 sum all the moredetails events in the specific session
- 5 Group by sessionId

Now that you've broken the sessions into events, you can take the next step and look into what happens in sessions where things are bought. One way to accomplish this is to reuse the preceding query, and add a filter to show only the sessions in which something was bought. Figure 3.7 shows the result.

But that doesn't get us to what we want. Because we want to know what interactions there were on each. Such that we can see what interactions were done before buying it.

3.2.5 Conversion path

A *conversion path* is the path that a user and specific content take together on their way to a buy conversion (and to live happily ever after). The exact definition is as follows:

The sequence of pages and actions a user takes from the time of arrival at the landing page (the first page that's seen) until the conversion.

This is different from a conversion funnel. A conversion funnel is made up of predefined goals that have to happen for the user to convert. A conversion path isn't just a linked list of events, but also includes a linked list of pages. Movie GEEK isn't the best website to describe this, as it provides only a limited set of events. Often a website will have a much longer list of events, including the following:

- View content
- View details
- Look inside
- Like content
- Share content
- Sign up for newsletter
- Search result click
- Campaign link
- Add to cart
- Add to favorites list (this will be added in the following chapter)
- Rate
- Write review

	sessionId character varying(200)	buy bigint	details bigint	moredetails bigint
1	7498	2	24	6
2	2851	2	21	2
3	2854	4	14	7
4	4132	2	6	1
5	5232	4	17	7
6	4111	1	3	3
7	4134	1	7	1

Figure 3.7 sessions where a buy event occurred.

- Buy

Therefore, a conversion path could be much more interesting than what MovieGEEKs will provide, as you have only five events. You're interested in the conversion path because events are often key indicators of where the relationship between the user and content is going (ironically, this analogy works great for dating sites). A key indicator could be that most users who use Search to find a movie end up buying it. You'd be able to interpret that event as something that pushes the implicit rating up an extra notch. How can you calculate these paths?

For each user, you want all the sessions in which a buy has happened; and for each buy, you want a count of the events that occurred on the item that was bought. Trying to come up with this information in my head makes me dizzy, but let's move slowly and see if we can do it.

First you find all the buy events for each user, session, and content, as shown in the following listing. Let's assume that everything that's done with a specific item is done before it's bought. This might be cheating, but it saves us from a lot of complex calculations.

Listing 3.4: SQL finding the happy couples of users and content

```
select session_id, user_id, content_id
from collector_log
where event = 'buy'
```

This gives you a list that you can then use to join up with the original table, so that you get only the events that happened in a session where that user did something with content that ended up being bought. This brings you to the next query.

Listing 3.5: SQL finding events leading up to a buy

```
select log.*
from (
    select session_id, content_id
    from collector_log
    where event = 'buy') conversions
JOIN collector_log log
ON conversions.session_id = log.session_id
and conversions.content_id = log.content_id
```

This will result in the output shown in figure 3.8.

id	created	user_id	content_id	event	session_id
40094	2017-07-03 17:10:45+02	3	2096673	buy	794776
40372	2017-07-03 17:10:45+02	1	1489889	addToList	885444
40367	2017-07-03 17:10:45+02	1	1489889	genreView	885444
40360	2017-07-03 17:10:45+02	1	1489889	buy	885444
40376	2017-07-03 17:10:45+02	6	1489889	buy	42456
40323	2017-07-03 17:10:45+02	6	1489889	moredetails	42456
40615	2017-07-03 17:10:46+02	1	1291150	buy	885445
40543	2017-07-03 17:10:46+02	1	1291150	moredetails	885445
40710	2017-07-03 17:10:46+02	6	1985949	buy	42462
40806	2017-07-03 17:10:46+02	6	1489889	buy	42463
40751	2017-07-03 17:10:46+02	6	1489889	details	42463
40724	2017-07-03 17:10:46+02	6	1489889	details	42463
40715	2017-07-03 17:10:46+02	6	1489889	details	42463
41613	2017-07-03 17:10:49+02	2	2120120	genreView	403980
41234	2017-07-03 17:10:48+02	2	2120120	genreView	403980
41228	2017-07-03 17:10:48+02	2	2120120	buy	403980
41276	2017-07-03 17:10:48+02	1	3110958	buy	885463
41386	2017-07-03 17:10:48+02	1	1083452	buy	885466

Figure 3.8 Snippet of the log

But you don't really need the buy events, as you know they're there. So the final query looks like the following listing.

Listing 3.6: SQL to find path conversion

```
select log.session_id, log.user_id, log.content_id
from (
    select session_id, content_id
    from collector_log
    where event = 'buy') conversions
JOIN collector_log log
ON conversions.session_id = log.session_id
and conversions.content_id = log.content_id
where log.event not like 'buy'
order by user_id, content_id, event
```

①

① the line that says we should disregard buy events.

You're not interested in chronological order right now, so this query provides the needed details.

3.3 Personas

Personas—the cornerstone of user-centered design and marketing—are fictive people, created to represent different stereotypes that correspond to groups, or segments in your user community. This section presents some personas; these are not a product of web analysis, but rather were created to span an area of the content in the MovieGeeks data (and in this case people who, incidentally, volunteered to be in my book, but normally personas are fictive people). The personas are used throughout the book exactly as marketers would use them. Later, you can look at the results of your algorithms and verify that the results correspond to their type.

So, without further ado, meet your new best friends for the duration of this book.

<p>Sara. Comedy, action, drama</p> 	<p>Jesper. Comedy, drama, action</p> 
<p>There is always room for another romantic comedy, except for when I want to see CSI style series.</p>	<p>I am up for laughs and will choose a comedy most days, but will watch drama and rarely an action movie.</p>
<p>User Id: 400001</p> <p>Therese. Comedy</p>  <p>Anything that makes me laugh is a hit.</p> <p>User Id: 400003</p>	<p>User Id: 400002</p> <p>Helle. Action</p>  <p>Anything with Superheroes' and anything the blows up.</p> <p>User Id: 400004</p>

Pietro. <i>Drama</i>		Ekaterina. <i>Drama, action, comedy</i>	
The more complicated it is, the better is the drama.			Nothing beats drama, but can watch an action, and rarely a comedy.
User Id: 400005			User Id: 400006

Each of these personas has unique tastes. A way to quantify these preferences is to indicate the number of hours each persona will spend on each genre, out of 100 hours. Using this description, you can describe each user as a tuple containing a number for each genre. For example, you can represent Sara's tastes as 60 hours of comedy, 20 hours of action, and 20 hours of drama, or taste = (60, 20, 20). Now doing this for each user, you get the following table.

Table 3.1 Persona Preferences

	Action	Drama	Comedy
Sara	20	20	60
Jesper	20	30	50
Therese	10	0	90
Helle	90	0	10
Pietro	10	50	40
Ekaterina	30	60	10

Another way of illustrating tastes is to plot them in a diagram, as shown in figure 3.3.

An advantage of plotting tastes in this fashion is that it makes it easy to find similar tastes (or to see that you created two users with the same tastes, as was the case when I first made the graph).

Some companies go so far as to make posters describing the personas and to require all the features to be described based on one of these personas. This creates many odd scenarios, as you end up discussing what one persona would do in a specific scenario, as if everybody were best friends with the personas. (And worse, at seasonal parties, I've even heard people discussing who had an affair with whom.)

Armed with these personas and their tastes, you'll move on to autogenerate some evidence data that can be used. Generating data seems like cheating a bit, but in our case, it's

a good way to start out with a data application, as you know what data you are working with. But remember that you should never expect things to be exactly the same in the real world.

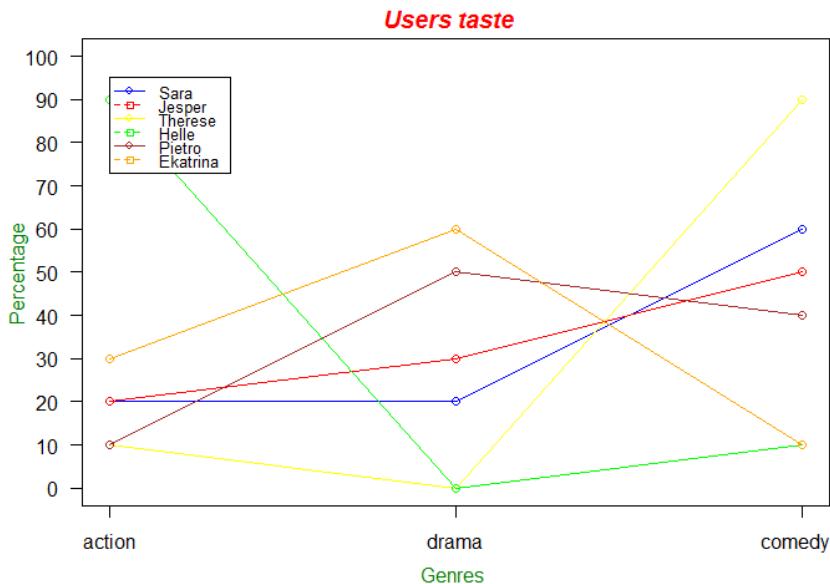


Figure 3.9 Chart of users tastes

The analytics part of the MovieGEEKs site also has a page for each user.

User Profile id: 4 (in cluster: Not in cluster)

Average rating: 7.83 / 10

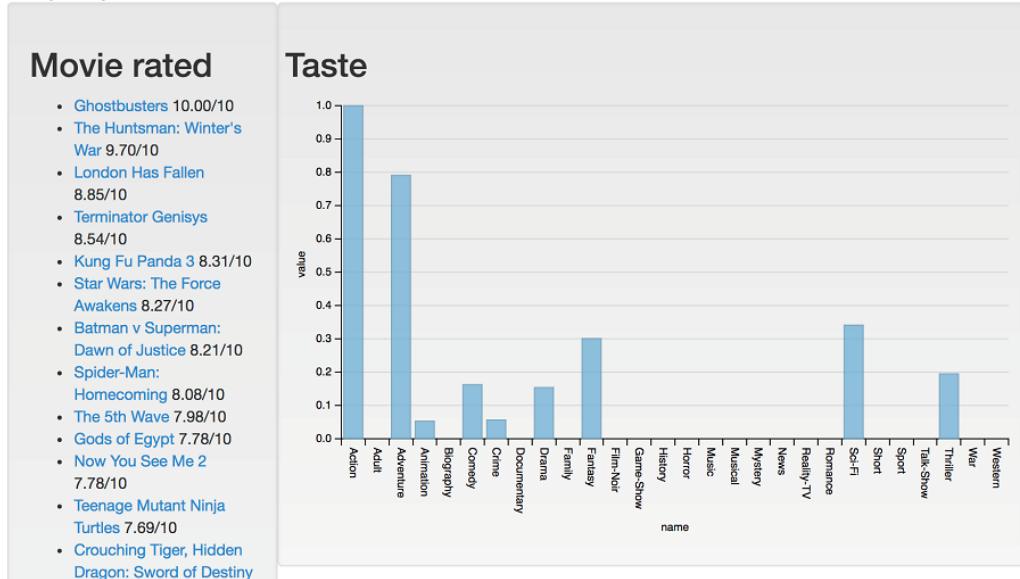


Figure 3.10 This is the profile of Helle above. She is very much into action. Here it also seems like she is into Adventure, but that is because many of the action films she likes are categorised both as Action and Adventure.

3.4 MovieGEEKs dashboard.

You can get the code of the MovieGEEKs site from github.

<https://github.com/practical-recommender-systems/moviegeek>

Either by cloning or download it. Please follow the readme.md for setup instructions. They have some data in database you should also run the populate_logs.py which I will give a short introduction to here

3.4.1 Autogenerating some data to our log

In this chapter and indeed the next couple of chapters we will be talking about implicit feedback, usually that is not something you can just download from the internet, but something you'd collect on your site. Since our site doesn't have many users, I have instead created a script which fills in some data, saving data into the Log database as we saw the collector also did.

To run it, execute the following:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/practical-recommender-systems>

Licensed to Asif Qamar <asif@asifqamar.com>

```
>Python populate_logs.py
```

The script will autogenerate logs for six users, which has specific tasks and will be used through out the book to show examples. These users or personas will be described in more detail in chapter 3. The core of the script is a for loop which iterates over the range from zero to the number of events wished for. At each iteration a random user is chosen, a film is selected according to the users taste, and then what action that should be logged. When all has been selected the event is saved.

Listing 3.7: opulate_logs.py

```
for x in range(0, number_of_events):
    randomuser_id = random.randint(0, len(users) - 1)
    user = users[randomuser_id]
    selected_film = select_film(user)
    action = select_action(user)
    if action == 'buy':
        user.events[user.sessionId].append(selected_film)
    print("user id " + str(user.userId) + " selects film " + str(selected_film) + " and " +
          action)

    l = Log(user_id=str(user.userId),
            content_id=selected_film,
            event=action,
            session_id=str(user.get_session_id()),
            #created=datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
            created="2017-04-03 12:00:00",
            visit_count=0)
    l.save()
```

The movieGEEKs dashboard can be accessed by running the site and going to the following url:

<http://localhost:8000/analytics/>

3.4.2 Specification and design of the analytics dashboard

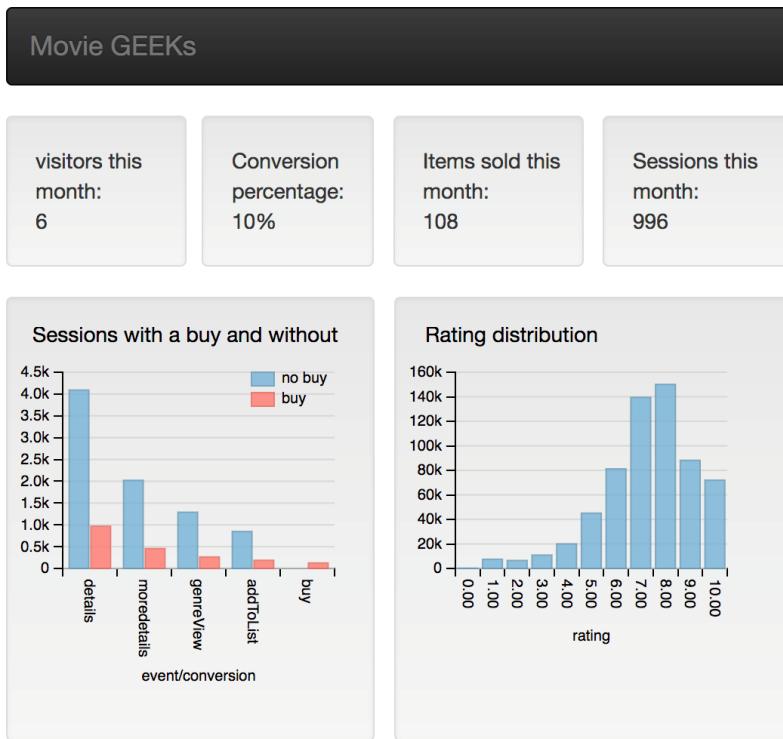
The analytics dashboard is something that most site owners dream about, and you'll try to make some of those dreams come true here. The site should contain a dashboard that fulfills the following requirements:

- Show how many visitors have come by, and what actions the users take
- Show how many visits result in buys (the conversion rate)
- Show the most sold products

3.4.3 Analytics dashboard wireframe

The dashboard looks like figure 3.9. The top of the dashboard contains the KPIs. They tell you the number of visitors that have dropped by within a period (since the log population script only work with 6 users, it will only be 6), currently the last month. The second component in

the top row shows the conversion rate, which is calculated from "how many sessions ended in a buy", then we listed number of items sold this month and last "the total number of sessions this month."



Top 10 content

- [Teenage Mutant Ninja Turtles: Out of the Shadows \(5\)](#)
- [Be Somebody \(4\)](#)
- [Now You See Me 2 \(4\)](#)
- [La La Land \(4\)](#)
- [Jackie \(3\)](#)
- [Logan \(3\)](#)
- [The Secret Life of Pets \(3\)](#)
- [Kung Fu Panda 3 \(3\)](#)
- [Snowden \(3\)](#)
- [Office Christmas Party \(3\)](#)

Clusters

Figure 3.11 Movie GEEK analytics dashboard

Below the gray boxes are two bar charts: the left one shows the number of times different events have occurred, and the right one shows the distribution of ratings. Finally, at the bottom is a list of the top 10 most bought content.

3.4.4 Architecture

The analytics app is another app in our Django project. Like all websites, it consists of two parts, front end and back end. Both parts have some work to do: the backend queries the database, and the front end visualizes the results of these queries.

The analytics app isn't something that should be accessible to the end user, but the security around that isn't something we worry about here.

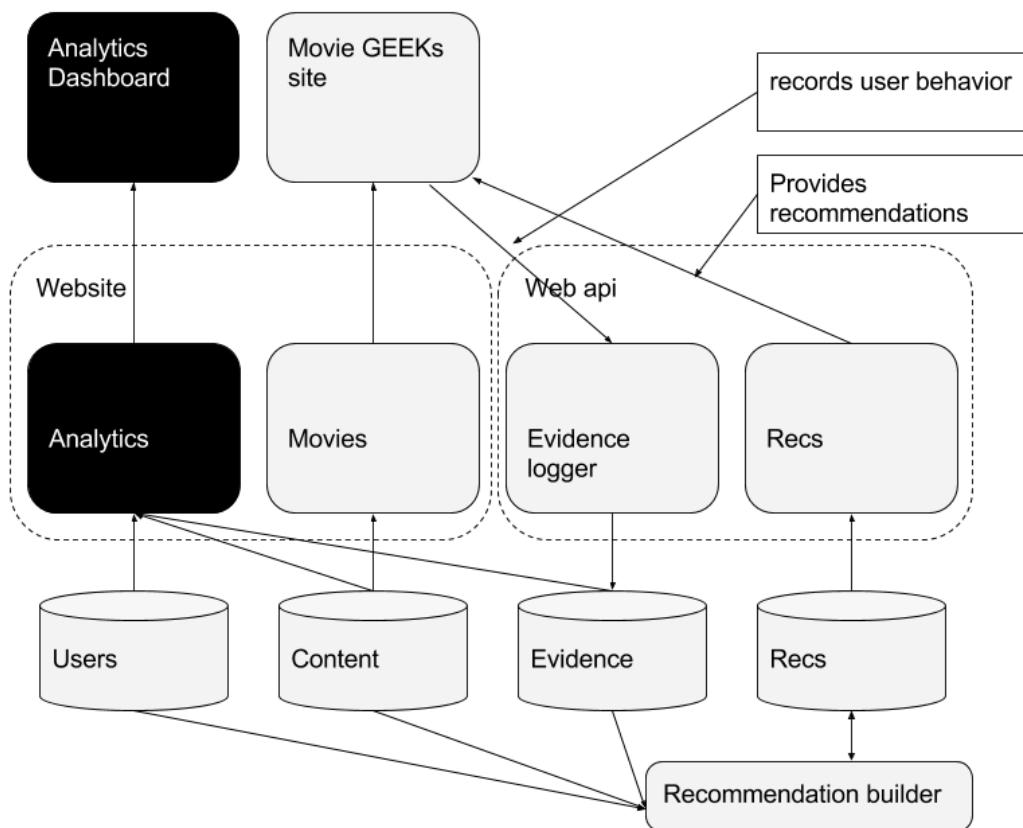


Figure 3.12 The architecture of MovieGEEKs with the Analytics parts of the system highlighted

Returning once more to our architecture diagram, the parts of the analytics app are marked with black in figure 3.10.

As you can see, we are keeping everything separate from the example site, which makes it easy to use the same architecture with any kind of site. This is considered best practice, and will enable you to use this for sites not implemented in Django.

The analytics app contains several views, but only a few views in a traditional sense, because they return a web page. The rest are used as a web API to retrieve data. The first traditional one, called a index, returns an HTML page like the one shown in figure 3.9. The json views return data and thereby feed the website with the data for each of its components. Figure 3.13 shows this setup.

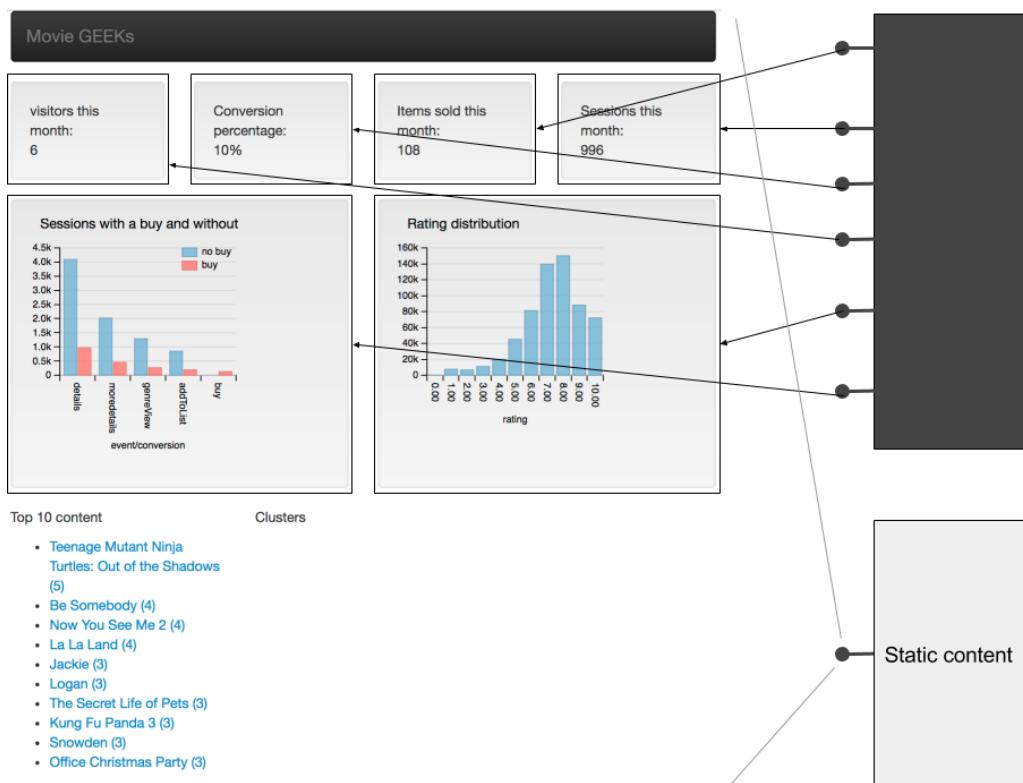


Figure 3.13 The static HTML is retrieved from the static content. Each component on the dashboard retrieves data from a different endpoint.

To understand what views, you need to implement, you should first take a look at each part of the dashboard and create a view method that returns data for each of them.

The top row, shown in figure 3.9, has four KPIs, which describe some numbers that are interesting to keep an eye on. The first one is the number of users who have dropped by your site. The next is the famous conversion rate. Then you have the number of products sold, and finally the number of unique visits (a unique visit means how many sessions, so the same user can have lots of visits).

A NOTE ON MONTHLY VIEWS

The KPIs are calculated for the last month, but could be daily or weekly, or even hourly if you have that much traffic. You can also use a sliding window to always calculate for the last month, or you can say this month, this week, and so forth.

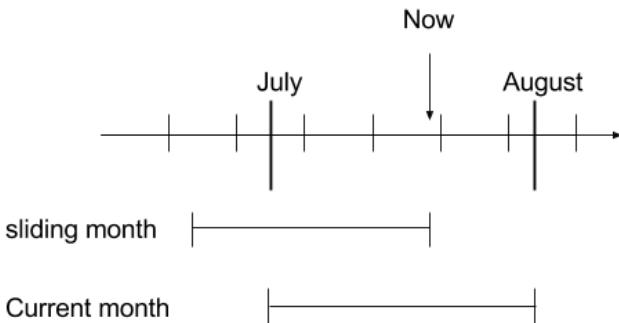


Figure 3.14 Different types of periods of time

A sliding window is illustrated in Figure 3.11. In a sense, what you do isn't terribly important, as long as it's consistent, so you can see if you are improving over time.

3.5 Summary and what's to come

In this chapter, you've looked at analytics, and how to implement a dashboard that shows simple information about how the website is performing. Having a Dashboard which enables you to understand how your site is doing will be a great help when doing recommender systems. You should now be familiar with the following:

- Key performance indicators are good, because they are things that can be benchmarked and easily used to see if your site is improving or not.
- A visitor is converts when it performs a goal and thereby does something that you are hoping for.
- Understanding your sites conversion funnel, is important such you can understand how close users are to convert, and helps you
- The conversion funnel shows a series of steps you would like the user to take. The conversion path, is the actual path visitors take before converting.

- Analytics is important to understand and have running always.

You're now finally ready to start looking at how to calculate ratings, and after that start creating non-personalized recommendations.

4

On ratings and how to calculate them

Hello, this is your persona speaking, proceed to learn the following things:

- You will be creating user-item matrices here
- Then you will revisit explicit ratings
- Having a firm grasp of why explicit ratings are not always good, you will dive into the mysterious implicit ratings
- To create implicit ratings, you will need to learn an implicit ratings function from which will translate evidence into ratings.

In this chapter, you'll transform your users' input to a format that you can use as input for the recommender algorithms. You'll start out looking at the user-item matrix, which is where most recommender algorithms start. Then you'll take another look at explicit ratings, the ratings that users add themselves. Implicit ratings are the core of our system, and you'll look at those next; first, you'll review what they are and then you'll learn how to calculate them from your evidence.

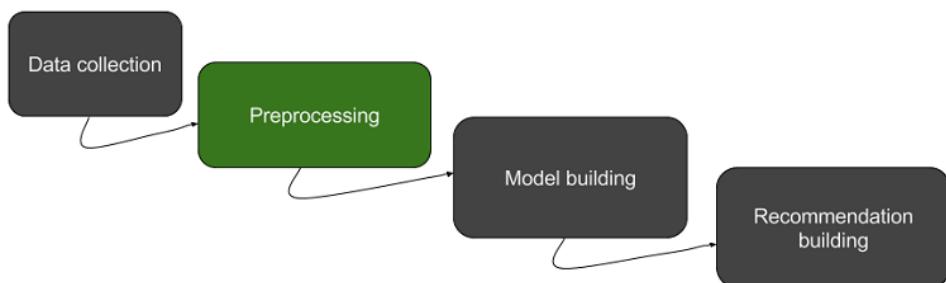


Figure 4.41 Data processing model for recommender systems

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/practical-recommender-systems>

Licensed to Asif Qamar <asif@asifqamar.com>

Figure 4.1 shows the flow of data we've talked about so far. Data collection happens when visitors interact with the site. Preprocessing is what you're going to do in this chapter. Model building and recommendation construction are handled in later chapters.

In this chapter, you'll convert web behavior to content ratings, which we'll use for the recommenders in later chapters. This type of ratings is called implicit ratings because they are deduced. Implicit ratings are used more and more because people seem to be unsure about what they like, and also tend to do (or watch) things that they'd otherwise tell their friends (or websites) that they don't like. I've watched films on Netflix that I then tell everybody who asks me that I hate, but I still watched them. Another good reason for using implicit ratings is that it is much easier to collect than the explicit ratings. That's why I'm a great fan of implicit ratings, which you've probably already noticed.

Having read the previous chapters, you should now have considered the following:

- What's the purpose of your site (the goals that you want users to achieve)?
- What events lead up to these goals?
- How many times have each of these events happened?

Keep those things in mind as you continue through this chapter. You're going to start by having a look at what most recommender algorithms expect as input: the user-item matrix. The idea of this chapter is to take the behavioral data and turn it into exactly such a matrix.

4.1 User-item preferences

A *user-item matrix* can be thought of as a table that has a row for each user and a column for each item (or the other way around). In literature, this is called a matrix, and we'll stick with that. In the junction between a user and a content item, a number indicates the user's sentiment towards the content item.

4.1.1 Definition of ratings

A *rating* is, for example, a number of stars on Amazon.com or Glassdoor (a site where people can rate their workplaces), or a list of hearts in my local newspaper's movie reviews.

Behind the scenes, a rating is a number on a scale—say, between 0 and 5—which can be translated into a graphical representation when shown to the end user. So, more formally, a rating is something that glues together three things: user, content, and the user's sentiment towards the content item, as shown in figure 4.2. Figure 4.2 shows what was saved after Jimmie had watched, liked and rated the Game of Thrones season 1, and therefore rated it four stars.

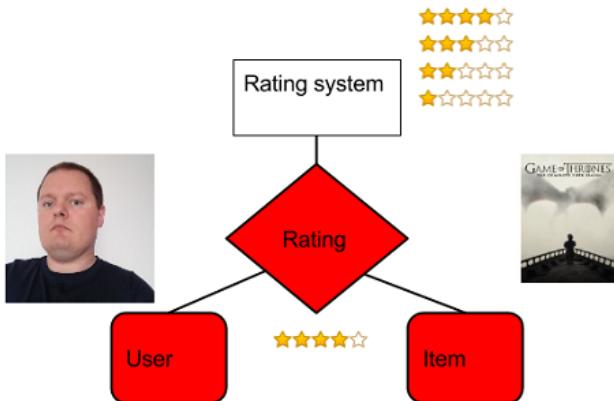


Figure 4.2 User content relationship

In a database, ratings are implemented as a junction table, connecting a user to a content item.

4.1.2 User-item matrix

An example of such a matrix is shown in table 4.1

Table 4.1 User-item matrix

	Indiana Jones	Micro cosmos	Avengers	Pete's Dragon
Sara	4	5		
Jesper	4		5	
Therese	5			3
Helle	4			5
Pietro		3	4	3
Ekaterina	3		3	3

An empty cell indicates that there has been no recorded interaction between the user and the item. Remember that there's a difference between an empty cell and a cell containing a zero; the latter represents the user giving a rating of zero, whereas the empty cell means that there has been no rating.

Those empty cells might not look like much, but they're the core of most traditional recommender systems. Most recommender systems attempt to predict what the user would put in them if they rated the corresponding items. Too few empty cells and the user has exhausted all content, too many and the recommender won't have enough data to understand what the user likes.

If you find yourself in conversations with people about the user-item matrix (which is something that occurs all the time, right?!), then a good topic to bring up is the sparsity problem. It's a bit like bringing up baby-feeding habits when talking to new parents; they'll light up and talk for hours. (New parents also speak of a sparsity problem, but a different one from the following.)

SPARSITY PROBLEM

Usually, a user-item matrix isn't as populated as the one shown in table 4.1. In fact, usually nonempty cells are rare, as many internet shops have lots of users and lots of items, but most users only buy one or only a few content items. So, you're much more likely to see a user-item matrix like the one in figure 4.3.

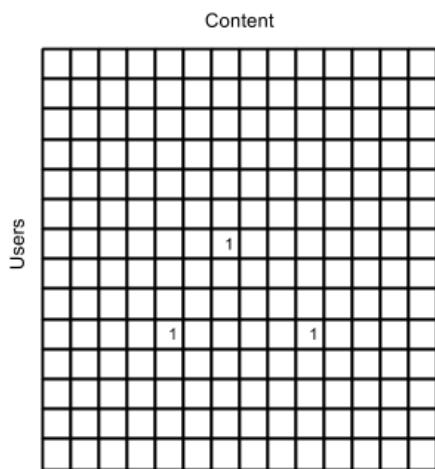


Figure 4.3 Sparsity table

The user-item matrix is input to the recommender algorithms that you'll see later. It's definitely not a good thing to have a matrix that looks like a Danish beach in wintertime (in case you never saw a Danish beach in winter, I can reveal that it's a lonely place to be). Figuring out how to add data into this matrix in the form of implicit ratings is the topic of this chapter.



Collaborative filtering use the user-item matrix to find similar users, so if the matrix is sparse (empty), it provides little information, and it is, therefore, difficult to calculate recommendations. In next chapter, you'll look at how to provide recommendations even if you have a sparse matrix. For now, let's focus on how to populate this matrix, either from explicit ratings (added manually by users) or implicit ratings (calculated from the evidence you collect).

4.2 Explicit or Implicit Ratings.

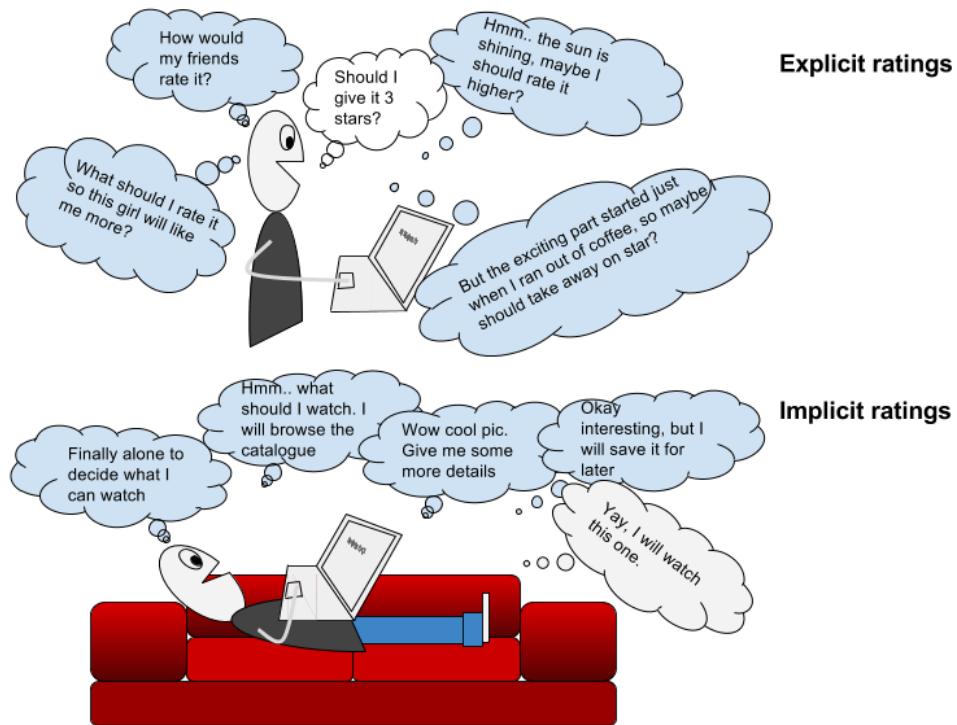


Figure 4.4 Difference between explicit and implicit ratings. Lots of things can influence what you rate and what you eventually watch.

The rating matrix shown so far in this chapter, the data inside are created to support the examples in this chapter. In our example app, we will get ratings from two sources, most

important source is the MovieTweetings dataset²⁰, the other part calculated from the user's behavior persisted in the data which is auto-generated in our case, as described in chapter 3. In a real app they could come from ratings added by users explicitly if such was implemented or created based on the users behaviour. Or they could come from a mix of the two.

On a movie site like MovieGEEKs, if a user bought something and then rated it, then the rating is probably trustworthy, or is it? I enjoyed films that I rated low, so what if the user buys two films very similar and rates them low? How much would we then trust the ratings? These are very domain-specific questions, so it is not easy to give general answers. But since the user keeps buying films that he rates low, then we should probably show more films that he would rate low also. On the other hand, a HBO user (another online streaming service) rates something that is only available on HBO, without seeing it, do we then trust it? Always have a critical eye on what data is showing you.

The truth about user's taste.

It is important to remember that the log data that we collect is evidence, and show an objective view of what the users does on the site. Translating it into ratings and opinions of a user is a very subjective process, which is something that has to be tweaked for each domain, but also for each recommender algorithm.

4.2.1 How we use trusted sources for recs

Are your fellow usertrustworthy sources for recommendations? Some places users are also selling things that mean that people will have an incentive to make themselves look better and make the competition look worse. For example, we could consider the following example. Somebody had gotten a crazy idea and wrote a book on recommender systems. And he was the boss of a big company (not all of this is true), and then he said all his 2181 employees had to give a positive review of his book, or they would get fired. Then the Amazon page could look like the shown in Figure 4.5figure 4.5. That is probably an extreme example, but I have seen it happen that a c-level person wrote a book, and then gave it as a present to all his employees afterward. Or people saying something is bad because the package was broken when it arrived in the mail. I am not saying that you shouldn't trust the reviews of other people just think about what incentive the users will have to either give good ratings or bad ratings.

²⁰ <https://github.com/sidooms/MovieTweetings>



Figure 4.5 Fake Amazon page of this book (I will leave it as an exercise for the reader to figure out which book pages I printscreened and cut together).

No matter whether it is explicit or implicit ratings, they can be faked, so remember that.

4.3 Revisiting explicit ratings

When a user manually gives a content item a rating, it is called an explicit rating. The easiest way for a system to populate the user-item matrix is to ask users to do it themselves, at least, in theory. Only they don't, even if you give them the chance. How many people review the books they buy or the movies they watch on Netflix? And even when people do, you can't always be sure that their ratings reveal their true opinions. People are influenced by what their circle of friends says. And what is being rated when a user rates? Try to think about that the next time you rate something: are you rating the complete package or was it just a detail that you didn't like that made you rate it low? Maybe you loved the film, but the DVD's cover was ugly, so you rated it low. Do you rate a good lawnmower low because of the way a screw is attached?

So, when you finish this book, hopefully it will inspire you not only to write recommender systems but also to spread the word that this is an excellent book. If so, then a way to do that is by giving it a high number of stars, for example, on Amazon (not trying to mentally imprint anything here—blink, blink). That's an explicit rating.

An explicit rating can be plotted directly²¹ on the matrix. So, if you have very vocal users, working with their explicit ratings is worthwhile.

EXPLICIT RATING BASED SITES

Websites such as TripAdvisor, Glassdoor, and others have based their sole existence on users' ratings. I mention this to remind you that even if implicit ratings generally show more-accurate opinions, explicit ratings still have their place in the world. We'll get back to explicit ratings later in the book, but in the next section you'll concentrate on the implicit ones.

4.4 What are implicit ratings

Implicit ratings are deduced from watching people's behavior. Sounds kinda scary when it's written like that. But remember, you're trying to ease information overload and help users, not stalk and manipulate them into buying more.

Most would agree that the event of a user buying a product indicates that the user has a positive opinion of the item; you can, therefore, deduce a positive rating between the user and the particular item. That's an implicit rating. The same is true when a user streams content or requests more information about a product. These are positive examples of a user-content relationship, whereas if a user returns an object, that's an example of an event that induces a negative implicit rating. To calculate implicit ratings is to take all the events recorded between each user and each content item and determine a number indicating how happy the user is with specific content. Another way of thinking about it is that you're trying to deduce the number of stars users would give content they've interacted with (viewed, streamed, bought, and so forth). You can define what each event means, but often the actions of users can be interpreted in several ways, so it's advisable that you create a system enabling you to easily tweak the calculations of ratings.

When the data is collected there are then many ways to use the data you've collected. Amazon's famous *item-to-item recommendation algorithm* uses only what users buy to create recommendations, but few can claim to have more than 200 million active users, as Amazon does²². Others use the browsing history of all users to recommend similar pages. Which of the two approaches better fits your site depends on what you're selling and what customers you have. After reading this chapter, you should have a better idea of which road to go down.

²¹ in on the matrix. Well, that's not completely true, often you normalize the ratings

²² Amazon for sure uses much more than the "buy" events, but in the particular algorithm described in *Greg Linden, Brent Smith, Jeremy York, Amazon.com Recommendations: Item-to-Item Collaborative Filtering, IEEE Internet Computing, v.7 n.1, p. 76-80, January 2003* they only use the "buy" events. 2003 is a long time ago.

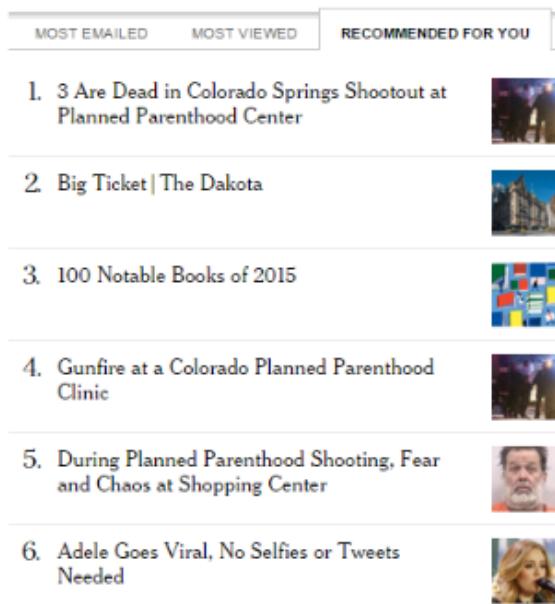


Figure 4.6 Recommended for you from www.nytimes.com

Some sites don't have the concept of "bought" at all. The New York Times website, for example, uses your browsing history to recommend you new material. Figure 4.6 shows the recs from the front page of the site (www.nytimes.com). The New York Times is forced to use implicit ratings, as the site doesn't allow users any explicit way to rate what they like. Moreover, even if it did let users rate articles, what would it mean if they rated an article low? Would it mean that they didn't like the topic, the way it was written, or just the specific story?

It's important to mention that even if a rating is high, that highly rated item might not be the best thing to recommend right now. A good example is that I love the mojitos made at a certain café I go to when I'm in Italy. That doesn't mean that I want one for breakfast in the morning, even if I've rated the mojitos from the previous evening very high several times in a row. Sometimes it's worth dividing the evidence into different timeslots and do different recommenders for each slot. But it is a trade-off between more accurate recommendations and data sparsity.

Always remember relevance, too.

Not all applications have a rating system, and it doesn't always make sense to add one. Another good example (besides the New York Times) is eBay, which sells unique items. For example, what information would eBay gain from you rating a rare 1984 Wonder Woman

lunchbox that you bought? eBay probably has only one to sell, but it's interested in knowing that you frequently browse comic collectibles and lunch boxes.

Many other sites have content that probably wouldn't benefit from ratings either, but they can still provide recommendations. These could be public sites with information documents, or real-estate sites, for example. User ratings also are hard to come by on educational course sites, which is therefore, another area where a lot of energy is spent on making implicit ratings.

4.4.1 People suggestions

A lot of CPU power is spent on calculating how to suggest people to other people. One of the more famous places to suggest people is good old LinkedIn, which also claims to be the first site to have done it (outside dating sites, I would venture). LinkedIn suggests "People You May Know," whom you should add to your network. This is an example of a site where it would be out of place for people to rate other people. But for sites such as LinkedIn or Facebook, some kind of calculation must be done to figure out what friends it should suggest to you.

We're walking on the edge here, and most people would say that we're leaving the realm of recommender systems and entering the realm of data mining.

4.4.2 Considerations of calculating ratings

In this section, you'll go through a few considerations of calculating ratings. Which approach to use depends on what kind of data is registered and what type of site the recommendations should be shown.

In chapter 2, you saw that a buy event is something that comes before the user rating it. The evil truth is that you don't know anything about this event, and that's the hardest problem to solve. But to make this work, you need to assume this:

A user buys an item because it looks good.

The item might turn out to be crap afterward, but generally, people buy things because they want them. The item might be a present for the user's mother-in-law, but the user bought something once, so why not recommend something else for the next present? So, let's go with that and look at the buy events in our evidence.

BINARY USER-ITEM MATRIX

Using the buy events, you can make a simple user-item matrix by using the following formula:

$$r_{ij} = \begin{cases} 1, & \text{when a user } i \text{ has bought item } j \\ 0, & \text{else} \end{cases}$$

Each cell in the user-item matrix will contain a 1 if the user bought the specific item, and 0 otherwise. A snippet of such a user-item matrix can be seen in table 4.2.

Table 4.2 Binary user-item matrix

Users	Movies			
	Indiana Jones	Micro cosmos	Avengers	Pete's Dragon
Sara	1	1	0	0
Jesper	1	0	1	0
Therese	1	1	0	1
Helle	1	0	0	1
Pietro	0	1	1	1
Ekaterina	1	0	1	1

A similar matrix could be generated by likes: 1 if the user “liked” the movie, and 0 otherwise.

The most known web shop that uses “bought or not” is Amazon. I regularly go to Amazon to look for books on Python or data analysis, but I often end up at the Manning or O'Reilly websites because I find that they allow more free access to the books, and because they are often half price. I've bought several books from Amazon that I can't read on my PC, only on tablets, that drives me insane.

Because I've done a lot of browsing on Amazon, Amazon should be able to see in my browsing history that I'm interested in Python and data analysis, but it uses only the books I've bought, as shown in figure 4.7. Recently I bought a list of books on Microsoft Azure, to a large extent because they were free.

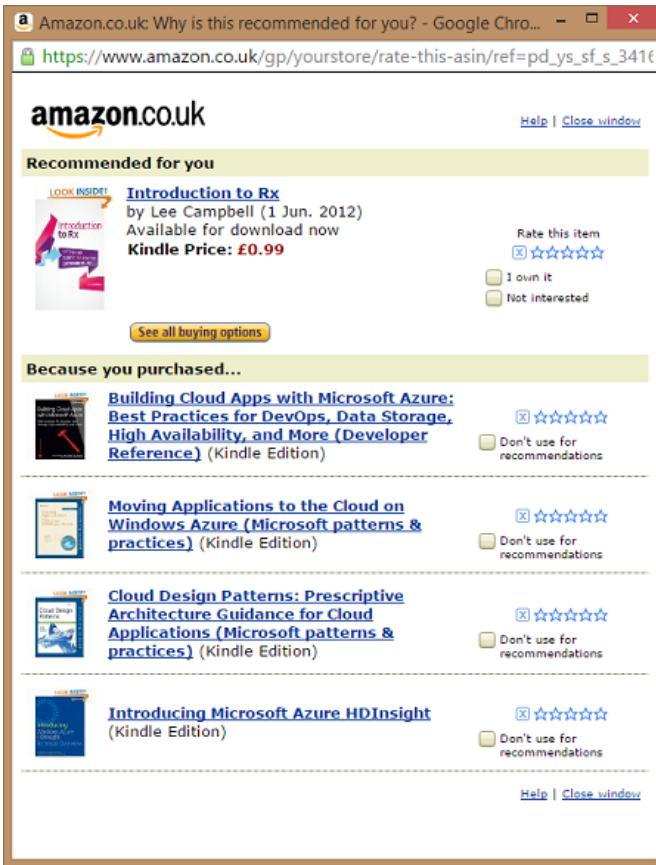


Figure 4.7 Amazon shows recommendations based only on what you buy

I bought my first book on Amazon around 2000 and have bought many since. But do they all represent my taste? Previously, I was a Java developer, but now that I'm over that phase, I'm not that interested in Java books any longer, so I hope that the things I bought recently are more important than the things I bought 15 years ago. In the next section, we'll look into that.

TIME-BASED APPROACH

Using a binary matrix makes all things black or white. But most websites would like have nuances, to enable the recommender system to get a better picture of what the users like.

There's a saying: *Nothing beats your first love*. But this isn't true in recommender systems. Here the most recent is given the most importance. Therefore, a way to make the matrix more nuanced is to use a function based on the purchase time. Some would go as far as to add the

production time of the item also. So a purchase completed 5 minutes ago of an item that was produced (or added to the catalog) 5 minutes ago will have a higher rating than a buy 5 minutes ago of an old product, but also of a buy last year of a product that was new at that time.

Using this approach favors new items. Even if a user buys lots of old romantic comedy movies, and only one new action movie, then the action genre will be winning, because old products get punished simply for being old.

Hacker News algorithm

Hacker News uses a somewhat similar algorithm, which puts lots of importance into recent events, and not so much on older ones. These types of algorithms are called *time-decay algorithms*.

Last time Hacker News published any details on how it works, it was using the following equation to calculate a news items ranking by taking the score (how many people upvoted this story – minus how many who down-voted it) and divided it with a time decay element. Using a term they call gravity, which indicates how fast an items ranking will decay:

$$\frac{\text{score} - 1}{(\text{Item age in hours} + 2)^{\text{Gravity}}}$$

The gravity is defined to be 1.8 (but the company tweaks this algorithm all the time).

People's tastes change over time, so what was the best thing at one moment in a user's life might not be the favorite now. This indicates it's a good idea to let old events count less than new ones.

BEHAVIOR-BASED APPROACH

Big companies such as Amazon would probably be drowning in data if they tried to use more actions than just the buy. But for most sites, the binary table shown previously would be quite empty or at least full of zeros. That's why it might be a good idea to broaden the horizon a bit and add other events than buy events.

Using several kinds of events requires a bit more thought, because you need to quantify how much a user shows liking via the different events. After the value of each event has been defined, each entry in the user item matrix can be calculated based on all the events that have occurred between the user and the item. This approach is what we'll work on, spicing it up with a bit of a time-based approach also.

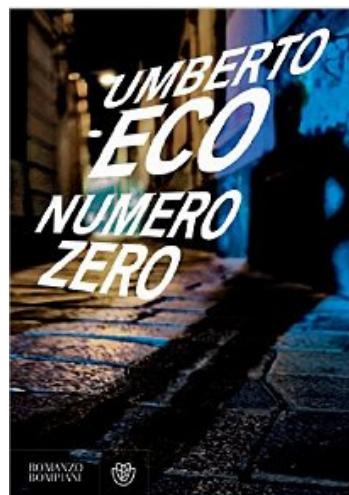
4.5 Calculating implicit ratings

Given the knowledge that you gained from working with the evidence in the earlier chapters, what can you say about the events? Can you say that events that led up to a buy mean that a customer was closer to buying? It's hard to tell whether that's always the case, but in most instances it's probably true. Therefore, you'll work through the events you're collecting on the MovieGEEKs site, starting with the buy event. Our example site is quite simple, with only Details and More Details events to calculate the implicit ratings, we're "lucky" that we don't have much complexity here. Most sites have many more event types.

It might be a good time to stop and think about what you're calculating, because in a sense the word "rating" isn't, actually, what you're trying to estimate, not even if you call it an implicit rating. Let's take an example.

My original objective for learning Italian was to read Umberto Eco in his original language. It was a bit of a silly project, because I never bothered to try to read any of his books even the ones translated, before setting out on going to evening classes in Italian. but it seemed like a good idea at the time. 10 years later, I'm now married to an Italian and speak Italian pretty well, although not well enough to make it enjoyable to read Eco's books. But that doesn't stop me from hurrying out to buy his books when a new one comes out, and try it, at least I did until he sadly passed away. Even if I'm not even sure whether I like his books in the end. On the other hand, there are all the writers that I do like and recommend to others. So, if you were implementing a book site, you'd want to know about all the books that I would potentially buy, regardless of how I would rate them. What you're calculating here is a number that will indicate how likely the user is to buy, not how high the user will rate the content.

Besides insights into my obsession with Eco books, this example should also have provided an idea of what you want to find now.



4.5.1 Looking at the behavioral data

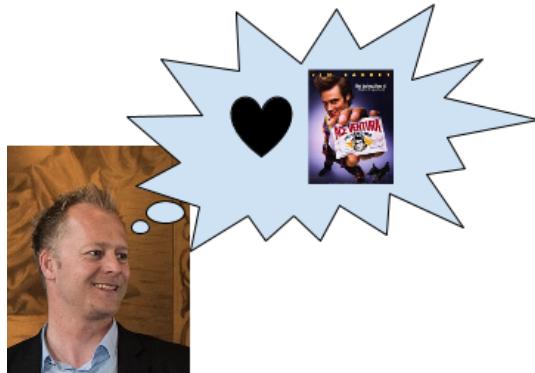


Figure 4.8 Jesper loves Ace Venture

Let's take an example from our Movie GEEKs site. Jesper (user 2) has a weak spot for Jim Carrey (illustrated in fig 4.8). He's considering buying *Ace Ventura*. So, he goes and looks at it once and thinks, ahh, maybe it's too expensive. Later he looks again, and then again. Finally, he decides to check out more details, which provide him with the final reason/excuse to buy it, so he buys it.

The list of events looks like the rows in table 4.3.

Table 4.3 Constructed evidence for Jesper

User ID	Content item	Event
2	Ace Ventura: When Nature Calls	Details
2	Ace Ventura: When Nature Calls	Details
2	Ace Ventura: When Nature Calls	Details
2	Ace Ventura: When Nature Calls	MoreDetails
2	Ace Ventura: When Nature Calls	Buy

So, you know that he bought it. But if I stopped the story before he made the transaction, you'd still agree that if a user looks at a content item three times, and clicks *More Details* once, then it's a positive thing.

On the other hand, if Pietro clicks *Ace Ventura* by mistake and never comes back to the page, then that action should not mean too much. But if he did come back, it would be more positive.

A thing that we left out of this example is the timestamp of the events. If clicks are done within a short time they might not mean as much as if the clicks has been registered over several days.

You could work through many such stories, but the core of the matter is that you can start adding some rules that your implicit recommendations should obey:

Buy => Top rating

One or more Details view + More Details => Very positive

Several Details views => Positive

One Details view => Not sure

In our dashboard, we made a diagram that shows which events most often lead up to a buy event. If you return to that, you also get a similar picture of how to calculate your implicit ratings. Figure 4.9 is a copy of a screenshot from chapter 3. In our case, we're cheating a bit: our chart shows auto-generated data, so the events distribute like this by design. But in a real system, that chart would be a good place to start looking at what value to attribute to each event.

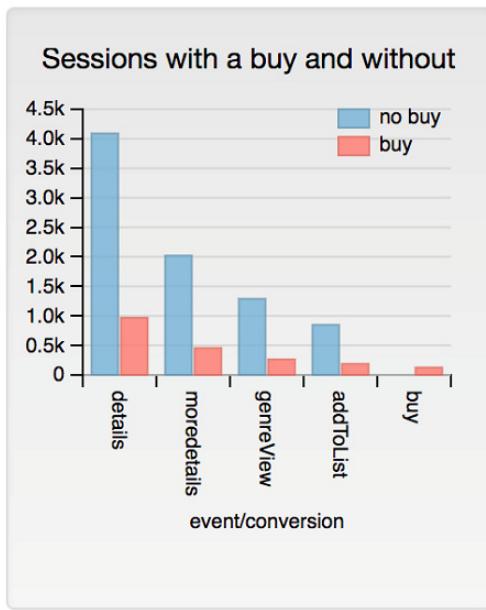


Figure 4.9 Chart showing how many events of each type have occurred, in sessions with a buy and without

To sum up this investigation, and with Jesper in mind, you can define an implicit rating function which outputs a number that shows how much a user u will be interested in buying

item i . To be more precise, you're interested in knowing how close a user u was to buy item i , so that you can use this knowledge to find similar things that the user might buy instead or also. With this in mind, consider the following:

Implicit rating of item i for user u

$$IR_{i,u} = (w_1 \times \#event_1) + (w_2 \times \#event_2) + \dots + (w_n \times \#event_n)$$

- $IR_{i,u}$ is the implicit rating.
- $\#event$ is the number of times that a specific event type has occurred.
- $w_1 \dots w_n$ are weights, which you will set based on the previous analysis (and probably tweak again when you start creating recommendations).

SHOULD ALL EVENTS HAVE A POSITIVE WEIGHT

If events occur in more sessions that include a conversion event than in sessions that don't, you can give those events a positive score. All interactions between user and content on our site and, in general, give a positive indication of the user being interested in the content, and should be treated as such. There are exceptions to be found, such as a Dislike button (which we haven't added to the site), but Netflix did²³, moving away from the star ratings, using just a thumbs up and down. The thumb down can be used to indicate that the user doesn't want to see similar content, which can only be interpreted as adding a negative score to the content.

CALCULATING WEIGHTS

So let's try to create a function by adding the weights according to what we learned in the example with Jesper and Ace Ventura above. As I've iterated several times before, it's important not to think that you can start out doing this once and then it will always work; it's a good idea to work with the weights and the function when you have a full recommender up and running and see how the result looks. First define the following:

```
Event1 := "buy"
Event2 := "moredetails"
Event3 := "details"
```

Now let's try to deduce the weights for this function, so that a buy has the maximum rating. Let's say the top rating is 100, and we can normalize it afterward. As listed previously, table 4.4 shows our assumptions and who this could be translated into weights:

²³ <https://www.nytimes.com/2017/03/31/opinion/now-netflix-is-all-thumbs.html>

Table 4.4 weights on events, where w_1 is the weight for buy event, w_2 is for more details view and finally w_3 is the weight of details event.

Events	Interpretation	Example of value
Buy	Top rating	$100 = (w_1 \times 1)$
One or more Details views + More Details	Very positive	$80 < (w_2 \times 1) + (w_3 \times 3)$
Several Details view	Positive	$50 < (w_3 \times 3)$
One Details view	Not sure	$(w_3 \times 1) < 50$

Having a list of equations looks like an optimization problem, which can be solved mathematically, but the equations have come from some assumptions, so you can't trust them as gospel. A way to solve satisfy the equations is to come up with w 's, which will obey the preceding rules:

$$W_1 = 100$$

$$W_2 = 50$$

$$W_3 = 15$$

If you try to insert these weights into the preceding function, it will look like the following:

$$IR_{i,u} = (100 * \#event_1) + (50 * \#event_2) + (15 * \#event_3)$$

As a small test, let's try to calculate Jesper's implicit rating of Ace Venture

$$IR_{i,u} = (100 * 1) + (50 * 1) + (15 * 3)$$

$$IR_{i,u} = 195$$

It will more or less provide you with what you want. If we should calculate an implicit rating for a film Jesper just found slightly interesting and clicked details, then more details it would be.:

$$IR_{i,u} = (100 * 0) + (50 * 1) + (15 * 1) = 65$$

We will normalize these ratings, which means that we will adjust them so they are between 1 and 10 later.

A thing to consider would be whether there is a cut off where the number of times somebody triggers an event doesn't add more information. For example, if somebody looks at details of some content more than a certain number of times. Think about it, if a user clicks on details 3 times on the same content item, does it mean that he is more interested in it than if he had clicked 10 times. It could therefore be worth to replace $\#event_n$ with $\min(\#event_n, relevant_max_n)$. The $relevant_max_n$ would be different for each event type. What

is says it that it will return the number of times the event has occurred unless it gets higher than `relevant_max_n`.

USING MORE AND MORE RELEVANT DATA

The first time a user returns, our ratings will be based mostly on previous browsing data, and not on buy data, so the recs will be based on few things. As the user interacts with the site, and hopefully starts buying more things, the content with which the user has interacted will narrow down to the users specific tastes.

4.5.2 This could be considered a machine-learning problem

Today many companies are spending a lot of energy trying to predict which users are ready to buy, and when. You have a similar problem here: you need to predict how likely a customer is to buy a specific product that the user has looked at. It can't be said for certain, that a relationship exists between a user's interactions with the site and what the user will buy, but it seems plausible. If so, some numbers or a vector could be multiplied with the data collected and produce a probability indicating how close a customer is to buying something. This is a fair approximation of what you're trying to calculate. If you defined that in machine-learning lingo, you'd say that there exists a function like the following:

$$Y = f(X) + \epsilon$$

Y represents the true prediction of a user's closeness to buying a specific item. In theory, this can be calculated by inserting your features (the data that you've recorded with the evidence logger). To save face, you include a parameter called noise, ϵ . ϵ is an indication that you can't calculate the complete truth from the features, so there will also be some sense of uncertainty, no matter how close your function f comes to calculating the real rating. The purpose of many machine-learning algorithms is to use data to approximate the function f , and I encourage you to try it out with your data.

So what kind of machine learning could you apply to this problem? You want to predict an implicit rating based on various events. To do that, you can only try to predict whether a series of events are leading to a buy or not. So you'll be looking at classifiers. A good classifier to start out with is the Naïve Bayes classifier. This classifier will provide a classification, but also a probability of how sure it is about the classification. In this case, you could use the probability that the classification is a buy, and use that as an implicit rating. To get more detail about how to use a Naïve Bayes classifier, see chapter 5 of *Reactive Machine Learning Systems* by Jeff Smith (Manning, 2017).

If you are only here to do machine learning, then please read on, because some of the algorithms used to calculate recommendations are machine learning. And those will be handled in detail in this book.

So let's see how we could implement implicit rating calculations.

4.6 How to implement these calculations implicit ratings

Enough beating around the bushes—let's get some code on the table! Let's start out with a quick overview of where we would want to implement this functionality in the MovieGEEKs site, and then move on to explaining how we go about it. More precisely we will look at:

- Retrieving data
- Calculating ratings
- View and understand.

BUT FIRST THE MOVIEGEEK SITE

To follow the following, it will be better if you had the MovieGEEK site running on your machine. It can be downloaded at from Github, from the following link.

<https://github.com/practical-recommender-systems/moviegeek>

Please refer to the instructions on the site, on how to install it.

1: RETREIVING DATA

To calculate implicit ratings for a specific user, you need to retrieve the log data from the user. Basically what you want is the data to tell you for each content item, how and how many times has the user interacted with the content. For the example shown above with Jesper, we want a row like the following:

User ID	Content ID	Details	Moredetail	Buy
2	Ace Ventura	3	1	1

Having this data will enable us to calculate the implicit rating. Getting data like that can either be done just by retrieving all data from the log containing the specific user id and content id or we can make the database work a bit more for us and return data in the format shown in the table above, which is what is done in the following SQL.

Listing 4.1: SQL script to retrieve ratings for a user

```

SELECT
    user_id,
    content_id,
    mov.title,
    count(case when event = 'buy' then 1 end) as buys, 1
    count(case when event = 'details' then 1 end) as details, 2
    count(case when event = 'moredetails' then 1 end) as moredetails 3
FROM evidenceCollector_log log
JOIN movies mov 4
ON log.content_id = mov.id 5
WHERE user_id = '4005' 6
group by user_id, content_id, mov.title 7
order by buys desc, details desc, moredetails desc

```

- 1 Count how many copies a specific user bought of a specific item
- 2 Count how many times a specific user viewed details of a specific item
- 3 Count how many times a specific user viewed more details of a specific item
- 4 Join with the movie table to get the table
- 5 Compare movie id with the content_id of the evidence
- 6 Filter on user id so you get the data of only one user, here its user '4005'
- 7 Use group by to enable the counts in (A, B, C)
- 8 Order things by buys, then details, and finally moredetails event

Having this data in hand (well... in memory) we can now start calculating the implicit ratings, which should be saved in the ratings database. The function implemented is as we showed above, and can be found in the code in the following method. Have a look at the following file, and find the method `query_aggregated_log_data_for_user` to see the actual code which retrieves the data: [prs/moviegeek/Builder/ImplicitRatingsCalculator.py](#)

2: CALCULATING RATINGS

The calculations are quite simple and doesn't require much explanation. We load the data from the database and calculate the rating. The rating is calculated using the weights which we deduced above. This method will be called for each user. The dataset called MovieTweetings, which we will use later in this book, has ratings on the scale from 1 to 10, so we will normalize these ratings to the same scale.

Listing 4.2: builder\implicit_ratings_calculator.py

```
def calculate_implicit_ratings_for_user(userid, conn=connect_to_db()):
    data = query_aggregated_log_data_for_user(userid)
    agg_data = dict()
    maxrating = 0

    for row in data:
        content_id = str(row['content_id'])
        if content_id not in agg_data .keys():
            agg_data[content_id] = defaultdict(int)

        agg_data[content_id][row['event']] = row['count']

    ratings = dict()
    for k, v in agg_data .items():

        rating = w1 * v['buy'] + w2 * v['details'] + w3 * v['moredetails'] ④
        maxrating = max(maxrating, rating) ⑤
        ratings[k] = rating

    for content_id in ratings.keys():
        ratings[content_id] = 10 * ratings[content_id] / maxrating ⑥

    return ratings ⑦
```

- 1 Call method which queries the database.
- 2 Create a dict to contain the ratings.
- 3 Iterate through each content item.

- 4 Calculate the implicit rating for content item.
- 5 Keep track of what is the highest rating so far
- 6 Go through all ratings, divide with max to normalize, and multiply with 10 to put it on a 0-10 scale.
- 7 Return ratings.

Having the implicit ratings on this scale also means that we would be able to use them in place of explicit ratings.

3: VIEW RESULT

If you fire up the MovieGeek app you can now and run the following:

1. Run `python populate_logs.py`

It will add auto-generated logs into the database. Flip back to chapter 3 for more info on this script. The database will now contain data shown in figure

2. Run `python -m builder.implicit_ratings_calculator`

It will calculate the implicit ratings.

3. Run `python manager.py runserver 8001`

It will start the moviegeek site, running on port 8001

<code>id</code>	<code>created</code>	<code>user_id</code>	<code>content_id</code>	<code>event</code>	<code>session_id</code>
100296	2017-08-14 22:04:06+02	400005	1355644	addToList	441008
100344	2017-08-14 22:04:06+02	400005	1355644	details	441008
100363	2017-08-14 22:04:06+02	400005	1355644	details	441008
100440	2017-08-14 22:04:06+02	400005	1355644	details	441009
100767	2017-08-14 22:04:06+02	400005	1355644	genreView	441014
100831	2017-08-14 22:04:06+02	400005	1355644	details	441014
100992	2017-08-14 22:04:06+02	400005	1355644	details	441019

Figure 4.11 A snippet of the auto-generated data. This is the data related to user 400005 and item 1355644

In the screenshot shown in figure 4.12 it is visible that two movies have received top rating of 10/10. If we look at the database (or just in figure 4.11) we see that there is actually no buy event, but it still got a top rating, that is due to the many interactions that has been done between the user and the content.

User Profile id: 400005 (in cluster: Not in cluster)

Average rating: 4.69 / 10

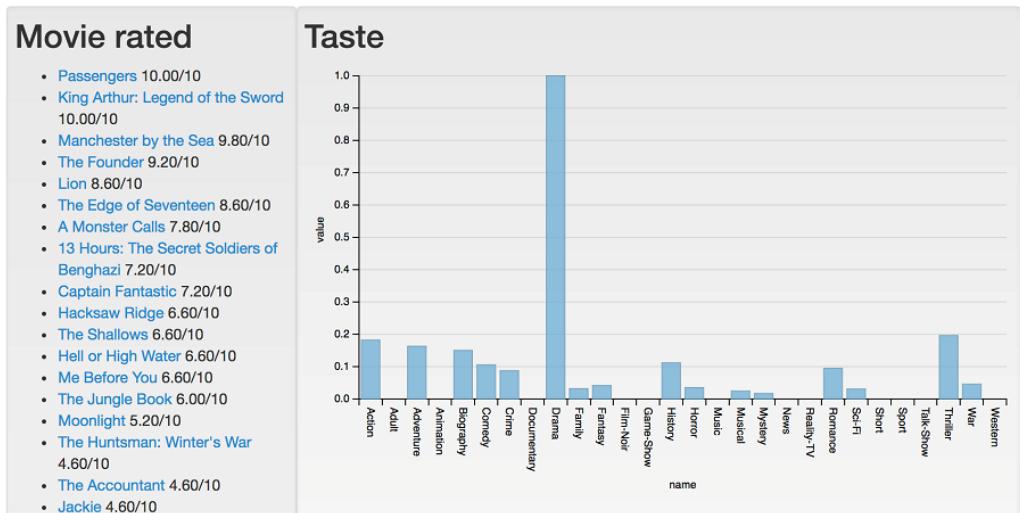


Figure 4.12 screenshot of the user profile of the user with id 400005. "Not in Cluster" means that the user is not part of any cluster, we will build clusters in chapter 7.

If you use implicit ratings like the ones we describe here, you need to take into account the the user hasn't consumed many of the content items which we have ratings for. As a result of this the items not bought could probably be included in recommendations, or maybe a user who has looked at and item as much as in this example, and still not bought it probably wont.

4.6.1 Adding the time aspect

If old behavior is less important than recent activity, it is worth adding that to the mix. The preceding implementation doesn't include any aspects of time decay. It makes it a bit more to include the time aspect also, as you need to look at each evidence point and add a multiplier, which becomes smaller as time goes by. Adding a time decay can be done in either SQL (in the DB) or code. Some companies swear to doing everything in SQL because they believe that's the best environment, whereas others say that SQL becomes unreadable with more than ten lines, so you'll have to move the code. The time decay function we are going to use the following formula:

$$score = \frac{1}{age\ of\ event\ in\ days}$$

There is no elegant way to do this in SQL, so we will do the decay in code instead. It also give you the opportunity to see how we can do everything in code, as oppose to above where we

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/practical-recommender-systems>

Licensed to Asif Qamar <asif@asifqamar.com>

aggregate the data first. This time we will just get all log data from the user using the following query:

Listing 4.3: Sql to a specific users log data²⁴

```
SELECT *
FROM collector_log log
WHERE user_id = {}
```

The reason for using days instead of seconds is twofold. First, you want all events that happened in one day to count the same because movies aren't something you'll buy several times in a day. Second, you want the ratings to decay slowly, so by using days, the weight of the events that happened a week before will have decayed only by a seventh. But a music-streaming site such as Spotify might want to give more importance to the last hour, or even the last 10 minutes. To illustrate how ratings decay with this function have a look at figure 4.7, you can see a plot of the decay with this function.

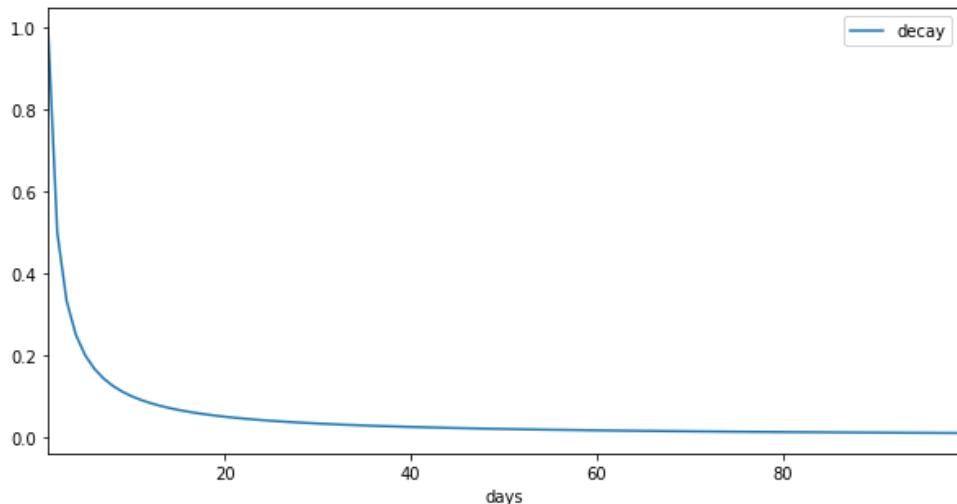


Figure 4.13 The function of the decay algorithm

To be sure you understand what it's happening, let's consider an example. If Jimmie buys *Game of Thrones* today, the event will have a score of 1; if the purchase was yesterday, the score is 1/2; a week ago is 1/7; a year ago is 1/365.

²⁴ If you have a large log, it might be a good idea to put a time constraint on it, like created > 1 month ago, and define an index on the columns user_id, created.

Listing 4.4: \Builder\CalculateImplicitRatings.py

```
def calculate_implicit_ratings_w_timedecay(userid, conn):
    data = query_log_data_for_user(userid, conn) ①
    weights = {'buy': w1, 'moredetails': w2, 'details': w3} ②
    ratings = dict() ③
    for entry in data:
        movie_id = entry.movie_id
        event_type = entry.event
        if movie_id in ratings:
            age = (date.today()-entry.created)//timedelta(days=365.2425) ⑤
            decay = calculate_decay(age) ⑥
            ratings[movie_id] += weights[event_type]*decay ⑦
    return ratings
```

- ① Add time decay to the buy events
- ② Add time decay to the details events
- ③ Add time decay to the moredetails events
- ④ Inner query to add the time decay

Lastly we should consider the calculate_decay method, which is shown in the following:

Listing 4.5: \Builder\CalculateImplicitRatings.py

```
def calculate_decay(age_in_days):
    return 1/age_in_days
```

It is left to the reader to try out more complicated algorithms to see if they improve anything. At this point of the book it is probably hard to understand what effects it has to change the decay function. But the concept is to add some idea of how relevant old things are to users. If you have a horse you are probably likely to not change taste in horse equipment ever so often, but if you are looking at a news site it is quite likely that old things won't interest you too much. For movies one day is quite small, and one month or year should be more likely, but for testing our system it makes more sense to use one day.

4.7 Less frequent items provide more value

Very popular items are often highlighted (very understandably) as good items to suggest to people but if we want to understand which things are important for people, then we actually need to look to the items which few users buy. Consider the following two examples:

- I buy the movie *Lord of the Rings*, which is number one on the charts (back then), and which every man and his dog have bought already.

- I buy a special collectors extended version, which includes a signed poster by the lead star, and which was produced in only 100 copies.

Which of these two events tells you more about my tastes? The first example will put me in a group with half the globe, since most people in the first example one doesn't make me unique compared to other users, whereas the other one puts me in an exclusive club with a maximum of a hundred members. And those hundred have a specific taste that's likely to be shared.

A different example often used to describe this problem is a grocery basket of bananas; everybody buys bananas, so knowing that the user buys bananas doesn't have much value, as compared to the specially imported sardines in chili oil, which is a special buy that says something about the buyer. So why am I mentioning this here. Well we could put a filter on the ratings we are calculating and boost the items which are special while the very normal ones would not get so high ratings.

To implement this can be a bit tricky, so let's take a quick stab at it. First, let's define a function that makes sense. This problem is closely related to the well-known *term frequency-inverse document frequency* problem (tf-idf among friends), which is often used by search engines as a tool to rank a document's value, given a user' query. You can consider this as a queryless search, where you want to attribute more value to the "special" items. And to understand what content items we shall consider special we will look for the inverse user frequency instead. The intuition is that if a user buys an item, the fact that it is bought by users who has bought few things makes I who buys many different things if everybody likes something, then knowing a user likes it, doesn't give us a lot of information about the user's taste, while if the user likes something only few people like, then it could be a better indication the personal taste of a user.

The function can be calculated in several ways. But the following shows one I find more interesting²⁵. To find the special items, you can calculate the inverse user frequency iuf, like this:

$$iif_{i,u} = \log \left(1 + \frac{N}{n} \right)$$

where

n is the number of times item i has been bought by user u .

N is the number of users in the catalogue.

²⁵ Its mentioned in regards of collaborative filtering in an article by J.S Breese called *Empirical Analysis of Predictive Algorithms for Collaborative Filtering* which I recommend you'd read.

Normalizing in this context means that we will put it on the logarithmic scale. Taking the logarithm of something is done to ensure that there is a big difference between the small numbers while the more numbers grow the less importance it has whether it's a thousand or two, this can be seen in figure 4.14.

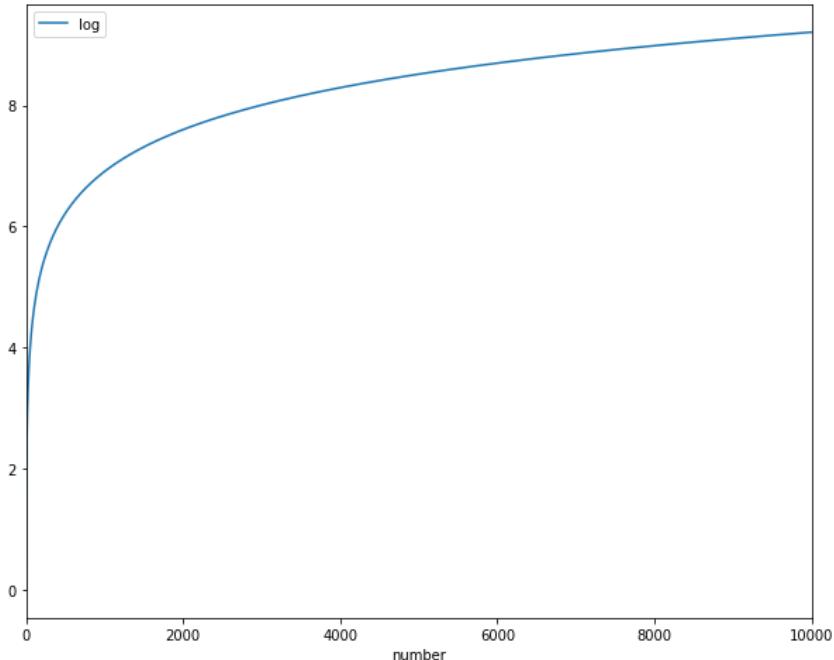


Figure 4.14 The log function on numbers from 1 to 10000. Numbers that go through the log function change a lot when they are small while when they become big the change gets smaller.

Since the number of users N is constant while calculating this, the iuf will have a graph looking like the one shown in figure 4.15

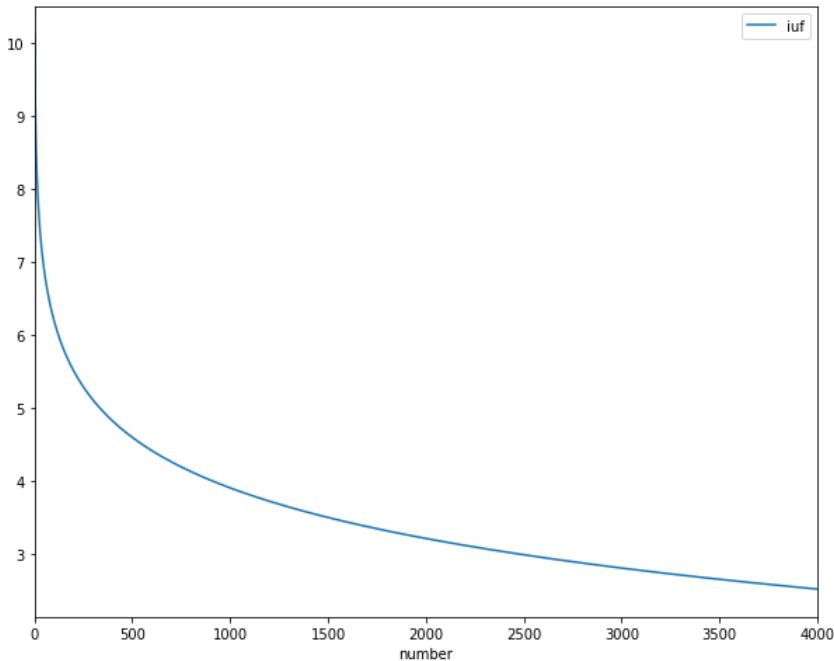


Figure 4.15 The inverse user frequency will look like this function. If only few people bought it then it will be boosted a lot while if many users has bought it, then it wont be boosted much.

The weighted rating calculated by multiplying the two is as follows:

$$wR_{ui} = R_{ui} * iuf_{i,u} = R_{ui} * \log\left(\frac{N}{1+n}\right)$$

If we want to keep these ratings between 1- 10 then we need to normalize again, as we did in code listing 2 above. It makes it easier to look at these ratings and compared them with the explicit ones.

This can be implemented both in SQL and on the server if you think that your site might increase conversions by adding this.

If we say that we are teaching machines how to predict ratings, then even if there is only one truth, then the role of a recommender system engineer or simply a data scientist, is as a guide. Not that I want to get all spiritual on you, but just remember that if the recommendations that come out in the end, seems to fit with the data, but not with the users, then one of the knobs to turn is here where you build the implicit ratings. The implicit ratings that you calculate are the foundation you create the recommenders on top. Do it well will give the recommender system the best conditions to predict well, do it badly and the recommender system will fail.

4.8 Summary.

Now that you have the dashboard and implicit ratings, you're well on your way to making your recommender system. The next step is to look at how to best save the trinity of user, content, and rating data. This is part of the next chapter, where you'll also see some recs. So go have a look! But first let's take a walk through the things you've learned in this chapter:

- A user-item matrix is the data format for recommender algorithms; you can populate them by using explicit as well as implicit ratings, or by indicating which items were consumed by the user, in a binary matrix.
- A rating is the glue that connects a user to an item; it can either be manually entered by the user or can be calculated based on the behavior of the user.
- The time-decay algorithm takes into account that not all information is equally important: old evidence is less important, as people tend to change their tastes.
- Inverse frequency factors into the equation, as interactions with less popular items provide more information about the user than interactions with popular items.

5

Non-personalized recommendations

We are recommending but not personal here. That doesn't mean that this will be less important:

- You'll learn that using non-personalized recommendations can also show interesting content. You see examples that shows why site should always order content and learn how to build charts to show users what is popular and highlight things other users have been interested in.
- You will learn how to calculate association rules, by creating item sets, based on shopping basket, and using that to create seeded recommendations
- Lastly you will see how the recommender component is implemented, which is the core component in the MovieGEEKs example site, and the one that provides the recommendations.

Non-personalized recommendations are usually where most sites start. It is easy to get started and does not require that you know anything specific about the users. Non-personalized recommendations are good because you are always able to show them, no matter how little you know about the users. Some would say that non-personalized recs should only be shown until the system knows enough about the user to show more personalized recs, but always remember that humans are flock animals by nature, so most will be suckers for knowing what content items are the most popular. If nothing else just to ensure what not to like.

We are handling charts and ordering, and association rules in this chapter. We will start out the party by looking at the good old charts, which everybody hates in these context-based days. Charts are simple recommendations based on statistics like which content sold more. Charts are about ordering your data, and it is, therefore, natural to continue with a talk about

ordering your presentation of the data. We will look at the implementation of chart and look at reordering the movies in the Movie GEEKs site. In the second part of the chapter, we will look at what people put in the shopping basket together and use that to create recommendations like “people who bought X also bought Y”, using something called item sets or frequency sets. The recommendations we will look at will be the same for all users interacting with the recommender system, that is why they are called non-personalized recommendations.

After five chapters, more or less entirely about how to get collect data about the users, you might think that it is a bit unfair to put the individual aside and look at the data as a whole. But remember most sites have lots of non-identified visitors, to whom you really want to cater because they are the future customers of your site. And furthermore even when you do know the identity of your visitor, it is highly likely that you don't have enough data to calculate personalized recs, and then it is good to fill out the blanks with some non-personalized ones.

5.1 What is a non-personalized recommendation

Back in chapter 1, we discussed the difference between a commercial and a recommendation. Let's briefly talk about that again.

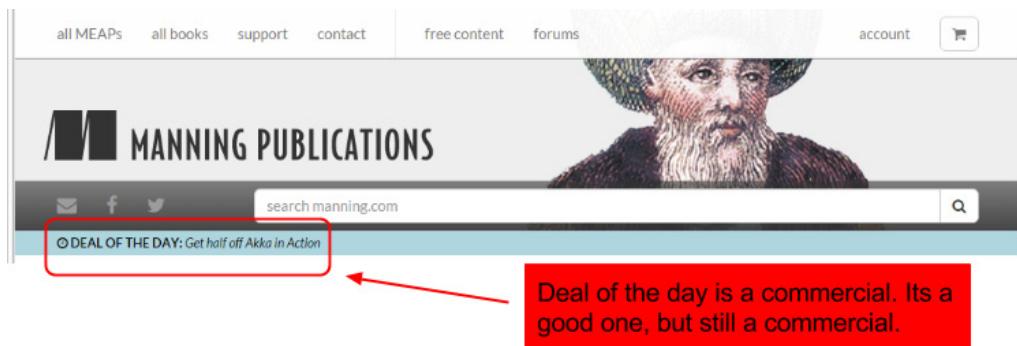


Figure 5.15 manning.com Deal of the day ad

5.1.1 What is a recommendation and what is a commercial.

The deal of the day from Manning, shown in Figure 5.1, is a commercial. It does not make it evil just because it is a commercial. A commercial is something that a vendor publishes because they want some content to be exposed to the users. And often people are interested in offers. However, it is something you should do with caution as bad commercials will drive visitors away. (I have an idea that the internet is full of such commercials, but after spending 30 minutes searching for one, I gave up. I did find a complaint about a guy who felt he got spammed after signing up on a paranormal dating site – It surprised him because he thought it was a serious site?!?). But, in the somewhat surprising lack of bad commercials, I would define a bad commercial as something that is blocking the user to get on with business, like a

popup that can't be closed before a film has been played, or redirecting the user to unintended pages.

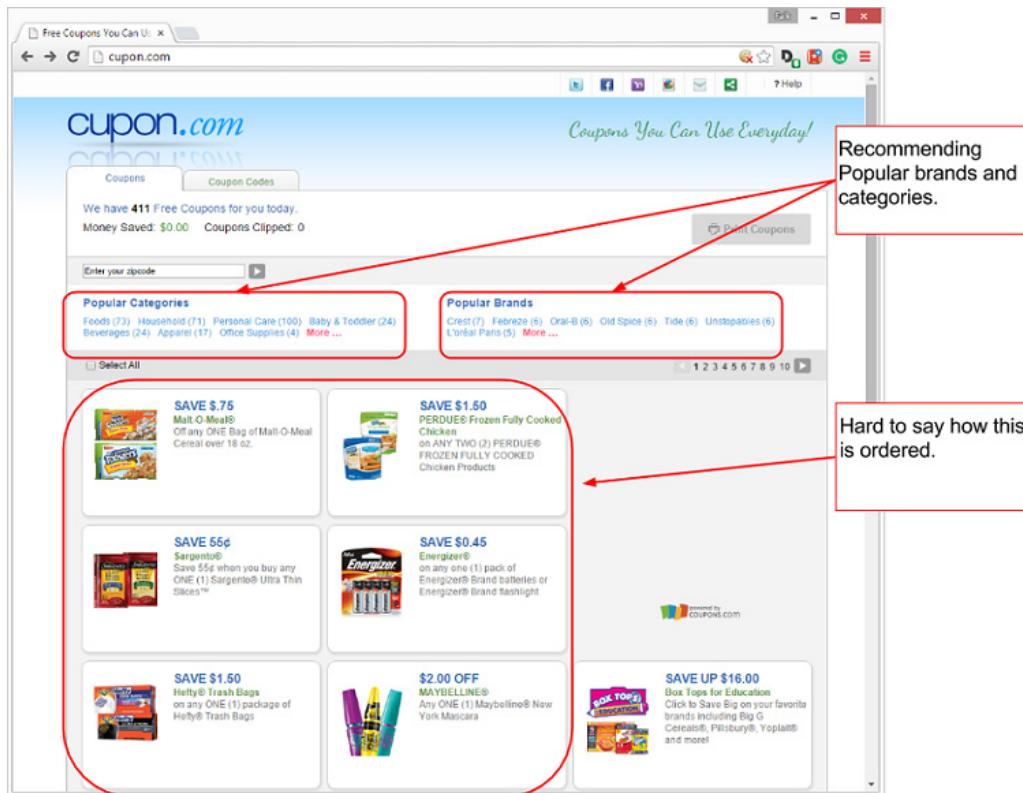


Figure 5.62 cupon.com collects coupons from everywhere

Often commercials are about things that the seller is trying to convince the user is a good offer or cheap (also if it often isn't). While a recommendation is about finding the user what they want. Of course, you could say that finding something cheap is exactly what the user wants. In fact, many sites have made a business out of recommending offers, of cheap things. Cupon.com is such a site, see Figure 5.2. Personally, I go searching for something after I discovered I needed it while a coupon site is a bit like people go searching for something they need, but didn't know it.

Cupon.com uses non-personalized recommendations to recommend more offers. In the top, it lists the popular categories and brands. While the central part of the screen contains lists of vouchers to save money on, that is hard to say how that is calculated. Cupon.com is

one of many choices, and I think it is a great way for sellers to get in contact with people, that are happy to spend more money in order to get things cheaper.

5.1.2 What is non-personalized recommendation

A recommendation, on the other hand, is something that based on data and calculated from data. To keep the definition from being to murky we will restrict it to be computer calculated based on usage data. That makes popular categories at cupon.com are recommendations (by calculating which category is more viewed).

Before we start calculating, let's look at some examples of what a site can do if they do not have any data at all.

5.2 How to make recommendations when you don't have any data.

We talked about it before, no data at all means no recommendations. A recommendation comes from what people like. So again, no data means no recommendations. What can you do then? You can fake it, by hand coding the recs to begin with (now we just defined that as commercials, but this is a more pure objective). For example you can call it spotlight like the do on Dzone.com, which is shown in figure 5.3.

The screenshot shows the DZone website with a navigation bar at the top. Below the navigation, there is a search bar and a login link. The main content area features a section titled "Big Data Spotlight" with a red arrow pointing to it from the left. This section contains two articles: "Cassandra to Kafka Data Pipeline (Part 2)" and "Paradise Papers: An In-Depth Graph Analysis". A red box highlights the second article, with the text "Editors are selecting content that they consider recommendable to the readers." A red arrow points from this text box to the article. Below the spotlight section, there are several other articles listed, each with a thumbnail, title, author, and download/save buttons.

Figure 5.3 DZone has editorial spotlights

However, if you do not want (or have the resources) to have somebody setting up a spotlight page, there are ways to do a little. First, what is it to present data without any order? A

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/practical-recommender-systems>

Licensed to Asif Qamar <asif@asifqamar.com>

"Local" example of that is the Movie GEEKs page, the films shown on the first page is simply the first items that came out of the database. But presenting data like that leaves a lot to coincidence. Much data could be inserted in the DB alphabetically, or if you always add data to the end, then it will always be the oldest shown.

Never use the order that came out of the database.

So always consider having a default ordering of the content presented.

Ordering by price is usually a bad idea.

But before doing that, just consider what that would mean. For cupon.com, that would mean the smallest items would always be on top. Another thing could be to order by biggest saving, but then all the small items would not be shown. So next step could be the percentage of the saving. So if you save 1 dollar on something that costs 10, it would be above something that you would save 2 dollars on but costs 100, since one is 10% while the other is 2% discount.

Using recency keeps the website dynamic

Thinking about our film website, we don't have any prices so what could be done there?

To start out one of the easiest way to recommend stuff is simply to order it according to what most people are most likely to like (assuming we know that). For example, if we order movies to show the movies most recent produced or content most recently updated. We will be making the site come alive and more dynamic, as long as the content is alive and dynamic.

Remember however that most recent things are not always the most desirable, if we were selling antiques we could probably work with ordering with the oldest first, like the prewarcar.com site shown in Figure 5.4. But in the case of the films, it is probably better to go with the newest ones first.

PREWARCAR

COKER TIRE WORLD'S LARGEST SUPPLIER OF ANTIQUE AND CLASSIC TIRES

CONCOURS BARNFINDS VETERAN BRASS VINTAGE BRITISH FRENCH GERMAN ITALIAN US made MOTORBIKES

Home Magazine Classics for Sale Parts & Stuff Featured My PreWarCars

Antique / Vintage / Pre-war Cars for Sale by Private Sellers

Start Prev 1 2 3 4 5 6 7 8 9 10 Next End

Car Make	Type	Body	Year	Loc	Price	advertiser
sort by ->	▲▼	▲▼	▲▼	▲▼	▲▼	▲▼
	Alvis Firefly	4 door tourer	1932	GB	GBP 20000	Private
Alvis Firefly 4-door tourer 1932 for sale						
Unusual, (possibly unique) body. Matching numbers, Part completed restoration, chassis and engine recently rebuilt. New aluminium body panelling, (painted), dashboard and steering wheel,. Requires fi						
	Delahaye 135 Competition	Disappearing Top Convertible,Figoni-Falaschi	1936	US	On Request	Private
Delahaye 135 Competition Disappearing Top Convertible Figoni-Falaschi 1936 for sale						
1936 Delahaye 135 Competition Disappearing Top Convertible Figoni et Falaschi, Paris Chassis No. 46864 Engine No. 46864						
	Austin YORK Light	SHOOTING BRAKE 16-6	1936	GB	GBP 18000	Private
Austin YORK Light 16-6 SHOOTING BRAKE 1936 for sale						
This lovely Shooting Brake was rebuilt approx 30 years ago by I understand a coach builder to quality yacht build standards and re-trimmed at the same time, she runs very well on all six and stops re						
	MG TA		1938	NL	On Request	Private
MG TA 1938 for sale						
1938 MG TA Built at the 21th of june 1938 3 former owners known British Racing Green Nice patina New fuel tank Blockley Tires in good condition Brooklands steeringwheel Full weather equipment						

Aston Martin
Audi
Austin
Bentley
BMW
Bugatti
Buick
Cadillac
Chevrolet
Citroën
Clément
Cord
De Dion-Bouton
Delage
Delahaye
Dodge
Donnet
Duesenberg
Fiat
Frazer Nash
Harley-Davidson mc
Horch
Invicta
Jaguar
Jaguar SS
Lagonda
Lancia
Lassalle
Mercedes-Benz
...

Figure 5.4 prewarcar.com

You can order the movies according to production time, but if it is a garden tool, then people don't really care much, well except if we are talking about Weber barbecue and Dane²⁶. If you are a male Dane, then it seems like you have to exchange your Weber barbecue at least once a year and it can only be bigger. That is due to genius marketing.

5.3 Top 10 - A chart of Items.

Back before the internet (yeah, I am that old), we had top 10s everywhere. There would be the weekly top 10 music chart in the radio in the weekends, which we religiously recorded on tape, and listened to on repeat all week. (Repeat meaning that you would listen to it to the

²⁶ Somebody from Denmark

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/practical-recommender-systems>

Licensed to Asif Qamar <asif@asifqamar.com>

end and the rewind, usually it would fill a whole cassette, but the side where the high numbers were, was never too good.) The top 10s were basically the only way we ever received recommendations, of course besides what we heard from friends. When MTV came to Denmark, it was insane...

Anyway, top 10 charts, like the ones shown in figure 5.5, have gotten a bad reputation since then, and that is a shame because they do show people what is popular, and no matter how much you are an individual and have a personal taste chances are that you like something on the top 10.

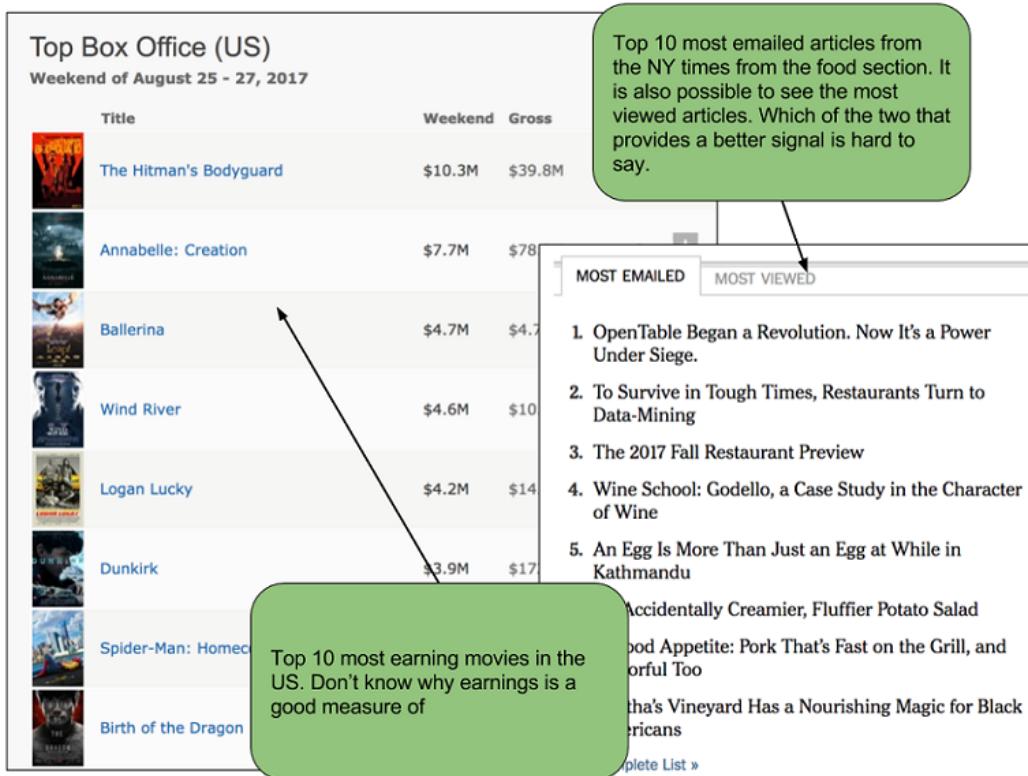


Figure 5.57 Top 10 most viewed best earning movies in the US according to IMDB and top most emailed articles in the food section of New York Times .com.

But then we are all sorted, right, top 10 is all we need, right?

Well not entirely, a top 10 chart will satisfy a large group (the majority even), but users are funny because they are not all alike. But this is great because the topic of recommender systems is much more fun when people are not alike. On a more serious note, it is also worth pondering a bit about what the top 10 actually tells you. If you have a number of users say 11

and you have 10 films. Then the most popular film might only be liked by two people leaving 9 others liking something else more.

Let's take a quick look at how an implementation of a chart could look, the chart simply says what content items was bought more times, like the one shown in Figure 5.6; we could also have done a chart on which items are viewed more, or to use a Facebook term liked more.

5.4 Implementing the chart and, in the process, the groundwork for the Recommender system component

We could add the chart quickly, as we did back in chapter 4, but to do things proper and add the functionality in the right place, we will start out creating the recommender system component, which is where we will do most of the work in the rest of the chapter.

5.4.1 The recommender system component

A recommender system can be architected in many ways, depending on how many recs you need to serve, and the size of content catalog and number of visitors. One thing is for certain is that you want it to be a structure which is independent from your website, as you can quickly drown performance. The solution that was done for the MovieGEEKs site is composed of two components (and a database). The first component, called builder, is used to do all the precalculations needed to serve the recs, while the second component is focused on the serving of the recs. The reason for the builder component is that most recommendations require a lot of calculation power, and requires time, which you don't have, when the user is waiting for the page to come back. It is therefore the goal of most recommender algorithms to precalculate as much as possible, to make the realtime performance as fast as possible. Normally you split recommender algorithms into memory based and model based recommender algorithms. The memory means that the recommender will access the log data at realtime, while model based signifies that the algorithm has done some kind of aggregation of the data, to make it faster to respond.

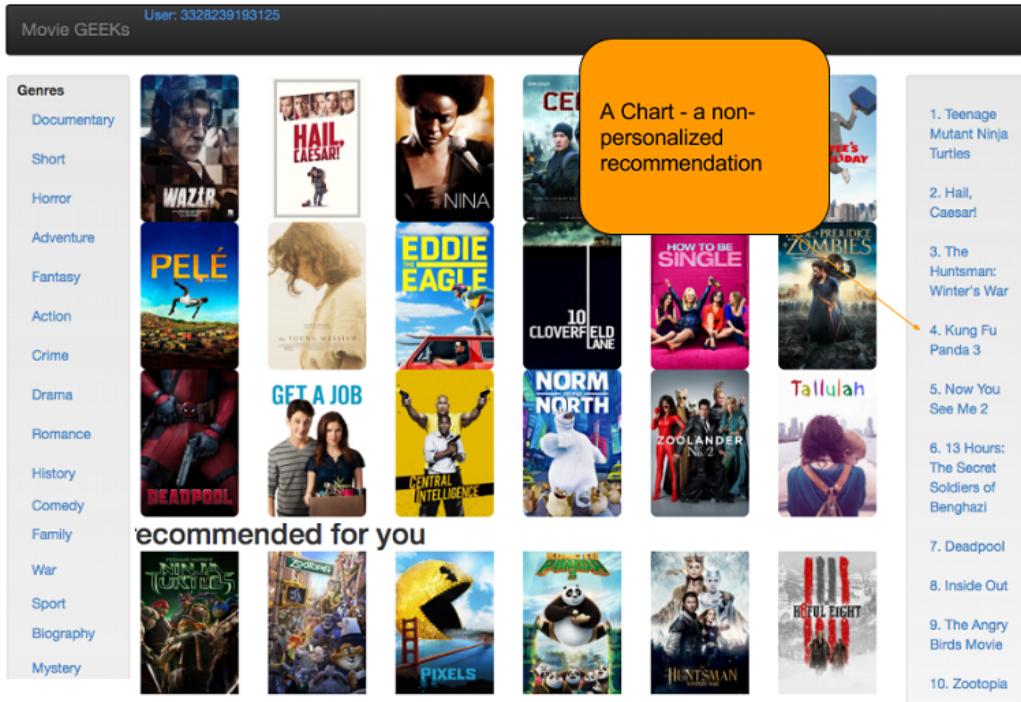


Figure 5.68 Movie GEEKS with a chart

Experience shows that memory based algorithms will only be working until a certain limit since it does not require many views per minutes before it becomes difficult for the servers to keep up with that.

For now, take a moment to enjoy the design in Figure 5.7. Always consider if you would do differently, and why? The light boxes contain the components we have already discussed, while the black boxes are the topic of this section. We have the Recommender (called Recs in the diagram). The Recommender app will handle requests from the website; we have the recs database, which contains that calculated recommendations (also called the model). The recommendation builder is the component that calculates the recommendations and saves them in the database.

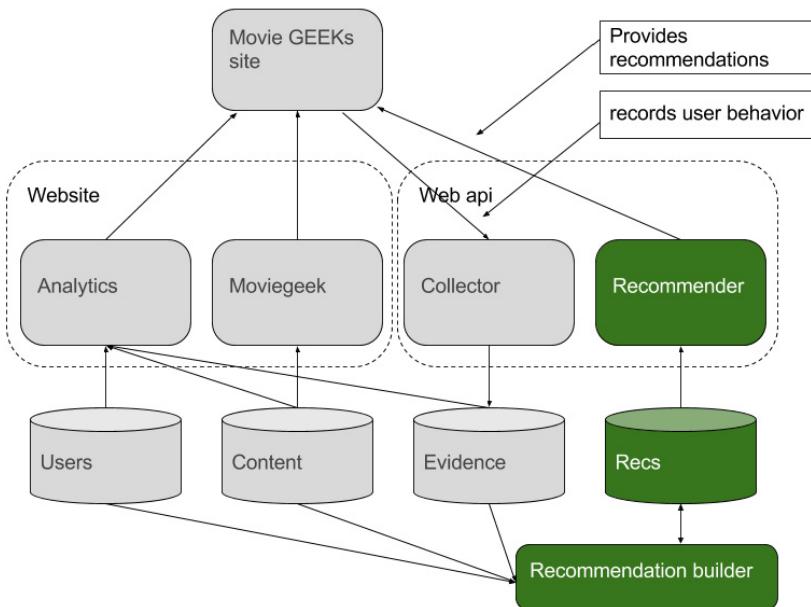


Figure 5.79 Movie Geeks architecture with the recommender system

5.4.2 Code from Github

Once again you should consider downloading the code from Github:

<https://github.com/practical-recommender-systems/moviegeek>

5.4.3 A recommender system

The recommender system is an app, which needs access to all of our data, and it needs to produce recommendations both non-personalized as well as personalized (albeit not before next chapter). I prefer the recommender system to be a separate part of a website, which can make it easier to scale. You should always keep the recommender system as a different part of your setup as it can be quite performance demanding, and you'd want to be able to separate it from other parts of the system. Often sites will build an architecture that will have a recommender system in a separate instance, and as a safety have a simple (even hard coded) fallback rec, which will keep providing data if the recommender system runs into problems.

5.4.4 Adding chart to Movie Geeks

A chart is a rather simple thing, basically we just need to count how many each content item has been bought and present that. With SQL it's a matter of simply grouping by content id and count the buy events in the log. The following sql query does one more thing, it adds the title of the movie to the chart:

Listing 5.1: Sql query to Get most sold products

```
SELECT content_id,
       mov.title,
       count(*) as sold
  FROM collector_log log
  JOIN moviegeeks_movie mov
  ON log.content_id = mov.movie_id
 WHERE event like 'buy'
 GROUP BY content_id, mov.title
 ORDER BY sold desc
```

- ① Get the content_id, movie title and the count
- ② Get buy count from the log table
- ③ Get Title from movie table
- ④ movie and log tables are connected using the movie id.
- ⑤ look only at buy events
- ⑥ group by the content id and title
- ⑦ and order by the sold column in descending order.

we are querying the log data. Which is empty if you just downloaded the code. But if you run the script found in the root called moviegeek/populate_logs.py, it will autogenerate some log data. This script is described in more detail in chapter 4. In a live system, it might be worth to calculate the chart ones a day, since table can be quite big and expensive to query every time a chart should be shown.

Being in the spirit of recommenders we will also say this is recommendations and hence be something that should come out of the recommender app. The chart method in recommender/views.py is as follows. It is not very nice code, but it works:

Listing 5.2: the chart method – recommender/view.py

```
def chart(request, take=10):
    sorted_items = PopularityBasedRecs().recommend_items_from_log(take)
    ids = [i['content_id'] for i in sorted_items]

    ms = {m['movie_id']: m['title'] for m in
          Movie.objects.filter(movie_id__in=ids).values('title', 'movie_id')}
    sorted_items = [{i['content_id']: {
        'title': ms[i['content_id']]}} for i in sorted_items]
    data = {
        'data': sorted_items
    }

    return JsonResponse(data, safe=False)
```

- 1 call the popularity recommender method to retrieve most bought items.
- 2 extract the movie_ids into a list
- 3 use the extracted movie ids to get the movie titles.
- 4 create a new sorted items list which contain the movie title also.

15 return as json.Listing 5.2: the chart method – recs/popularity_recommender.py

```
def recommend_items_from_log(self, num=6):
    items = Log.objects.values('content_id')
    items = items.filter(event='buy').annotate(Count('user_id'))
    sorted_items = sorted(items, key=lambda item: -float(item['user_id__count'])) ①
    return sorted_items[:num] ②
```

- 1 retrieve the items and count how often they were bought.
- 2 sort by number of users who bought it.

To see the output from the method try typing url (if you have the server running at port 8000): localhost:8000/recs/chart, I get a chart of the 10 most sold items, as shown in fig 5.6.

5.4.5 Making the content look more attractive

Let's look at the content, above we said that you cannot allow the database to dictate the order of how you present your content. So if we looked at MovieGEEKs site and let it show the movies as they are in the database, it would show you some really old films, like shown in figure 5.8

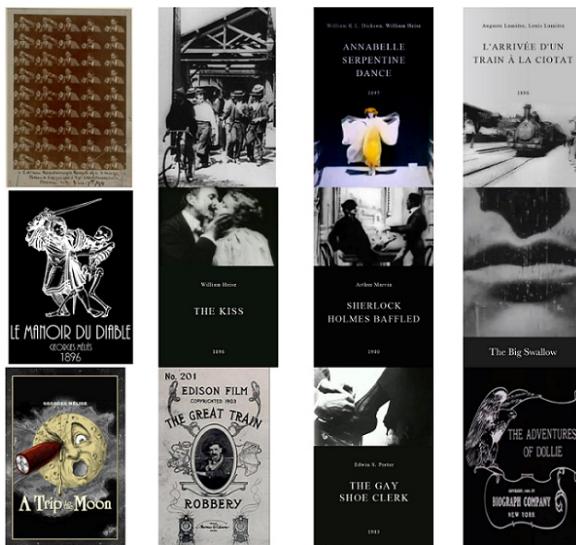


Figure 5.810 movies as they are ordered in the database.

You can play around with the ordering by finding the following line in the view file.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/practical-recommender-systems>

Licensed to Asif Qamar <asif@asifqamar.com>

Listing 5.2: moviegeeks/views.py – index method

```
movies = selected.movies.order_by('year')
```

The line above orders things by year, but will start at the lowest first, so if you have a thing for black and white films, this is great, but I think most people nowadays will want more recent ones. To change the direction of the order in Django query, you simply add a minus in front of the column name, like the following:

Listing 5.3: moviegeeks/views.py – index method

```
movies = selected.movies.order_by('-year')
```

Which makes the frontpage of MovieGeek a bit more interesting, as you can see in figure 5.9

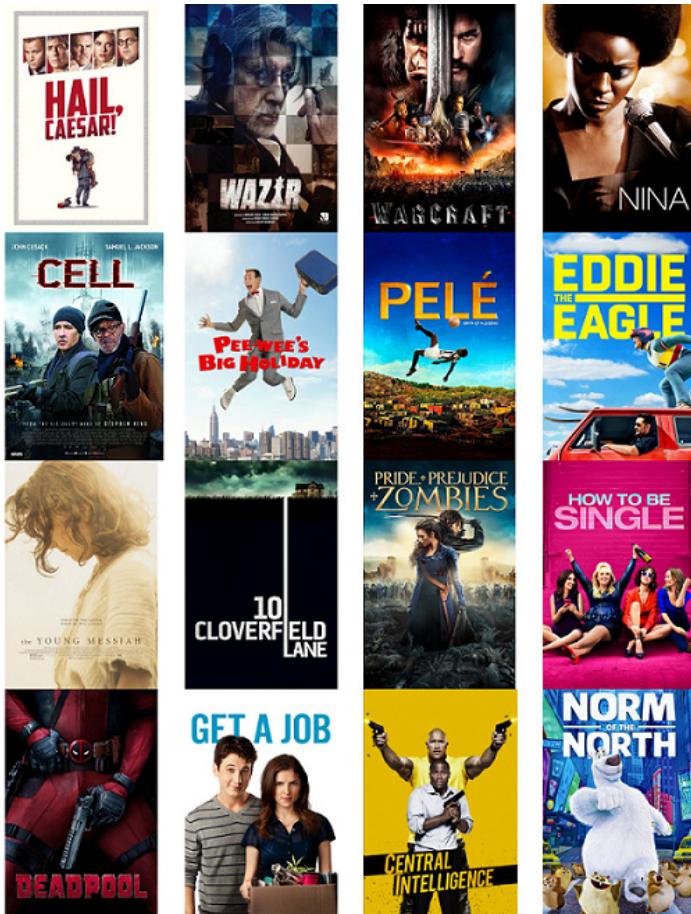


Figure 5.9 ordering films by year

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/practical-recommender-systems>

Licensed to Asif Qamar <asif@asifqamar.com>

You can order the movies according to production time, but that won't work for many things as customers are not always looking for the latest one. Turning again to gardening tools we might not get much out of ordering it by production date, but maybe there will be data about when the gardening tools are mostly used. So you use machines to prepare the ground in early fall, then later you need things to put down seeds, then a gadget to keep wheat away etc. In other words, they could be ordered in seasonal usage, with the current season in the top.

With ordering in place and some non-personalized recommendations (e.g. chart) it is now time to start looking at some more concrete recommendations.

5.5 Seeded recommendations

Charts are good but very generic, one thing that many sites take advantage of is the information that you are looking at a specific item, and use it to create recommended associated items. These items could be said to be seeds.

Seeded recommendations, is that not just a search? It is kinda, but the idea is that it is an item, product or article that you use as input and then you use to other relevant things. We will use items bought together to figure out how to suggestions. To understand know people buy together we will look in our evidence.

Frequently Bought Together

Total price: £43.49

Add all three to Basket

- This item: Frostfire Large 2 Person Instant Popup Tent £24.99
- New Set of 2 x 180cm Camping Yoga Roll Eva Foil Foam backed Sleeping Mat Mattress Tent Festival... £8.69
- Yellowstone Essential Mummy Sleeping Bag £9.81

Figure 5.10 Amazon.com Frequently Bought Together

One of the most famous seeded recommendations is the **Frequently Bought Together**, which is shown Figure 5.10 whenever you look at an item at Amazon, and almost every other webshop. The way to solve this problem called an affinity analysis or in more farmer terms basket analysis. So how could we go about doing that.

5.5.1 Top 10 items bought by same user as the one you are viewing.

Can we solve this just by finding all products that are bought together with the current product we have right now, and then take the top X of that? We could but as we will see below it doesn't work very well.

It turns out, that one of the challenges of showing this, is the fact that most products are bought together with the popular products. A classic example is most people who comes out of a supermarket in Denmark will have a liter of milk in the bag, so almost no matter what else is in the basket you could say that it was frequently bought together with a liter of milk.

In fact my wife just came home from the supermarket with not only one but two litres of milk (Figure 5.11 shows the shopping receipt). So most products in a supermarket that is bought often will also be bought often with milk. So there will be frequency sets containing milk and most other products.



Figure 5.11 What my wife bought tonight (The list contains: plastic bag, yogurt, milk, milk, and olives)

When two or more items are often seen together it is called a **frequency set**.

So, we need to figure out another way of doing it. You are probably thinking that yeah that's great, but this is only interesting if you are a supermarket. Nevertheless, this can be applicable in many areas, in the following we will first have a look at how to calculate frequently bought together products in a simple supermarket example, and then move on to implement it in the movie site. But beyond that, we can move to some larger things, very few estate sales websites or boat selling sites has frequently bought together, since that is a market that usually only sells one item at the time. But even if you do sell large things like that, then it is worth thinking about recommending smaller things. I never bought a boat, but I would venture that no matter what size the boat, then you need a life west with it. And depending whether it is a sailing boat or a speedboat you might need special equipment. So even when it is large and expensive things, it is worth adding frequently bought together, or rather most *frequently extra equipment*.

5.5.2 Association rules

Instead of looking at most popular objects, some clever people has come up with Association rules. You could think of an association rule as something a bit closer to kindness than the

marketing talk above. Association rules in the commerce scenario can be thought of as well-meaning advice. Most people hate to come home with a new harddisk only to realize that they do not have any cable to connect it with. If you buy things on Amazon then you are in luck because they remind you that most people buy it together with a cable. As you can see in fig 5.12.

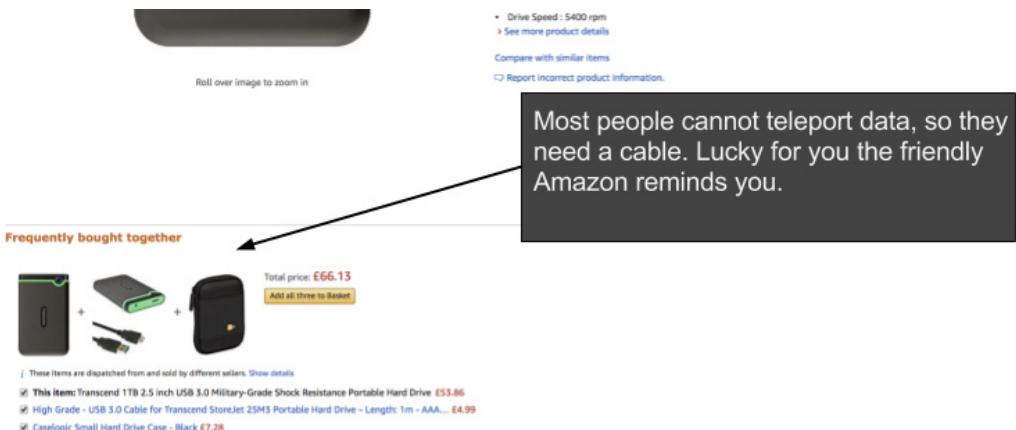


Figure 5.1211 Amazon.com External hard disks are frequently bought together with a cable

But let's think a bit about how to get these association rules. Below is a short list of supermarket checkouts. (I tried to add an example about Star Wars here, but I don't remember them ever doing their groceries, so we will stick to supermarkets instead). Let's imagine that we have a supermarket that only has five products: milk, dates, yogurt, carrots, and bread. Which will usually be called items when talking about association rules.

- 1: { bread, yogurt}
- 2: { milk, bread, carrots, }
- 3: { bread, carrots}
- 4: { bread, milk}
- 5: { milk, dates, carrots}
- 6: { milk, dates, yogurt, bread }

Each of the lines above is a transaction, which contains a number of items. To make an association rule, we can look at items that are bought together. If we pick milk, then all other products are bought together with milk. So does that offer us much value. No matter what product you have, you can say that other people bought milk together with the current product... so that is not a lot of values. Then what?

We need to find the products that are always bought together, but not bought with everything else. Any subset of the list of items above is called an itemset. Bread and milk are an itemset {Bread, milk}, and they can be found in 3 out of 6 transactions. So does that make us confident that it is a good idea to recommend milk when ever there is bread in the basket.

Let's try to define some numbers that can make it easier to decide if a rule is valid or merely by coincidence. The problem about the 3 out of 6 above is that bread is present in 5 out of 6 transactions, so it would be hard not to find transactions containing bread. To take that into account we will define *Confidence* as the number of transactions where the itemset is presently divided by the total number of times the first one is present.

DEFINITION:

Confidence

$$c(X \rightarrow Y) = \frac{|T(X \text{ AND } Y)|}{|T(X)|}$$

Where

$T(X)$ is the set of transactions, which contain X .

So, lets calculate what is the confidence that if there is bread in the basket at checkout, then there will also be milk. This can be written like this

$$c(\text{bread} \rightarrow \text{milk}) = \frac{|T(\text{bread AND milk})|}{|T(\text{bread})|}$$

Now we need to find all the transactions containing first both bread and milk and then only bread.

$$\begin{aligned} T(\text{bread AND milk}) &= \{\text{milk,bread,carrots}, \{\text{bread,milk}\}, \{\text{milk,dates,yogurt,bread}\} \\ T(\text{bread}) &= \{\text{milk,bread,carrots}\}, \{\text{bread,carrots}\}, \{\text{bread,milk}\}, \{\text{milk,dates,yogurt,bread,carrots}\} \end{aligned}$$

Inserting that into the equation we get:

$$\begin{aligned} c(\text{bread} \rightarrow \text{milk}) &= \frac{|T(\text{bread AND milk})|}{|T(\text{bread})|} \\ &= \frac{3}{5} \\ &= 0.6 \end{aligned}$$

So according to this calculation we would be 60% confident that you find milk also when you see that there is bread in the basket. That seems okay right. But wait a minute, if we do the same with dates and carrots, then doing the same calculation would give us

$$c(\text{dates} \rightarrow \text{carrots}) = 0.5$$

While I get that bread and milk often go together, I have a different feeling that of all datelers (people who eat dates) buy carrots half the time they buy dates. I do not think that the evidence support it. In the sense, there are not enough cases of transactions with dates to support the claim. That leads to a second definition; we use to understand whether there is an association rule between two items or not.

DEFINITION:

Support

$$S(X \rightarrow Y) = \frac{|T(X \text{ AND } Y)|}{T()} \quad \text{Where}$$

$T(X)$ is the set of transactions, which contain X .

$T()$ means all transactions.

Looking back at the two examples above give us the following support

$$S(milk \rightarrow bread) = \frac{3}{6}$$

$$S(dates \rightarrow carrots) = \frac{1}{6}$$

In other words, evidence that supports for the association rule $bread \rightarrow milk$ is much larger then for the association rule $dates \rightarrow carrots$. Which was also the conclusion above.

So, this is a nice little example. But if we zoom back to real life, then most shops (at least the ones that survive) have more than six products. And the transactions might well be much larger. When I started to write this chapter I asked my friends on Facebook what they had bought last time they went to the supermarket hoping to get some good example data, but the feedback was far too messy to work with. As shown in **Error! Reference source not found.**, which also illustrates that association rules can quickly become much more complicate to calculate.

Kim Falk
1 hr · København · [See Translation](#)

Hi,

I am writing a section on association rules (used to make the frequent bought together recommendations) and I need a list of what you bought last time you went to the supermarket. It could be great to get some data.

Thank you in advance.

[Like](#) [Comment](#) [Share](#)

[User] Grovboller, agurk, rød peberfrugt, skyr, koldskål, gulerodder, kyllingfilet, pastabånd med broccoli og ærter, 4 poser Haribo slik på tilbud 😊.
[Unlike](#) · [Reply](#) · [1](#) · 1 hr

[User] Pregnancy test, metal coat-hanger.
[Unlike](#) · [Reply](#) · [1](#) · 1 hr

Kim Falk Jorgensen Not sure that is going to add any association rules, but thanks 😊.
[Like](#) · [Reply](#) · [1](#) · 1 hr
[View more replies](#)

[User] Mozzarella and bacon on a quick trip after børnehaven tod 😊.
[Like](#) · [Reply](#) · [20 mins](#)

[User] Write a comment...

[User] [qualsiasi supermercato o solo In Danimarca? Se va bene anche dal Cile contribuisco volentieri](#)
[See Translation](#)
[Like](#) · [Reply](#) · September 17 at 4:49am

[User] [Va benissimo 😊](#)
[See Translation](#)
[Like](#) · [Reply](#) · September 17 at 10:18am

[User]
[Like](#) · [Reply](#) · [1](#) · September 17 at 3:44pm

[User] [Yesterday supermarket shopping:](#)

- Low fat milk
- Orange juice... See More

[Like](#) · [Reply](#) · [1](#) · September 17 at 7:23pm

Figure 5.13 Helpful friends on Facebook were quick to respond to my call for shopping data.

To find association rules we need first to find frequency sets. Figure 5.14 shows the possible frequency sets there could be if you have a set of items containing four elements {milk, butter, dates, bread}. Once again, this is a simple example, but to explain how the implementation is done we need a diagram like this.

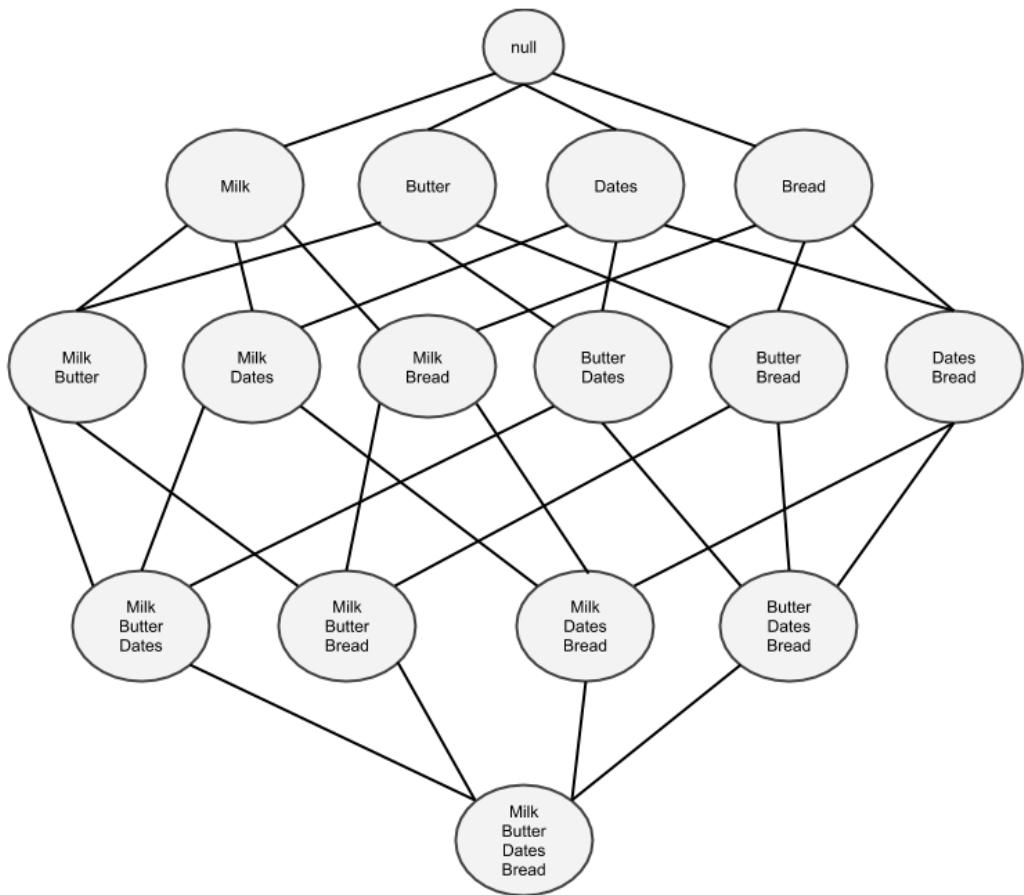


Figure 5.14 Itemset lattice with 4 elements

The diagram shows all the combinations of length one, two, three and four elements of the itemset. So, if we start from the bottom, where all elements are there, what needs to be fulfilled for that frequency set to exist. All elements need to occur often together in a transaction. But if all of them needs to be there, then each item of the itemsets should also be there frequent. If we go one step up and pick the node (the circle) all to the left in the lattice, we find an itemset {milk, butter, dates} shown in figure 5.15 also.

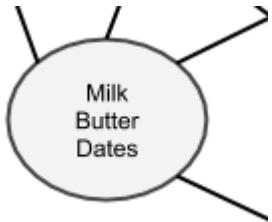


Figure 15.15: Itemset lattice, with low presence of butter

For this to become a frequency set, we know that milk, butter and Dates needs to occur often, and mostly together. But the fact that all of the items have to occur often than we can take advantage of that to make a fast implementation. That means that if we start out looking at itemsets with only one element to see if they are frequent. For example, if we find the butter is almost never present, then we know there are no frequencysets with butter. That means that we can cross off all rules that contain butter. The black nodes in Figure 5.16 shows which elements we can remove from the list just by checking butter and finding it infrequent. With that in mind, let's see how we might implement this.

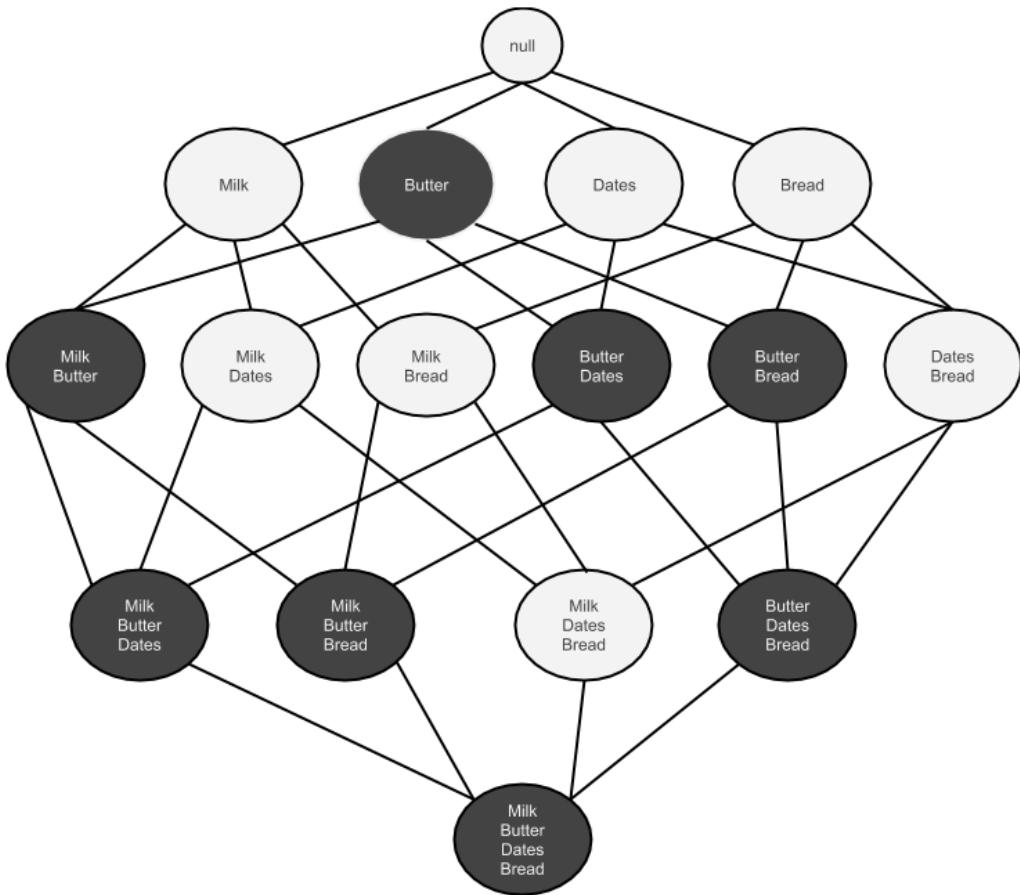


Figure 5.16 The black nodes in the diagram shows which frequency sets will never produce any association rules because butter is not frequent, then none of the nodes containing butter can be frequent.

5.5.3 Implementing association rules

The procedure we described above goes something like the following:

1. Settle on a minimum support and minimum confidence level.
2. Get all transactions
3. Create a list of itemsets, one for each element and calculate their *support* (number of times it is present, divided by the number of transactions) and set *confidence* to one.
4. Build a list of item sets (containing more than one item) and calculate support and confidence by
 - a. For each transaction:
 - i. finds all combinations of items and

- ii. add one to the item sets support.
- 5. Iterate through the item sets and remove the ones that do not fulfill the confidence requirement.

Let's translate this into Python code, but let's wait a bit setting the minimum support and confidence level. And just try to calculate everything to start with.

2: GET ALL TRANSACTIONS

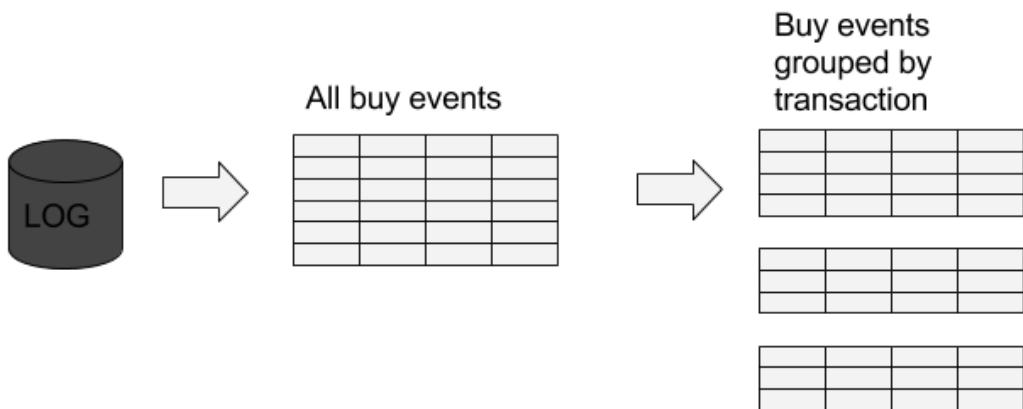


Figure 5.17 creating transactions from the log data.

On the MovieGeek site there is no concept of a basket, so we will instead say that buys happening in the same session is a transaction. We will get our transactions from the log, this means that we need to first retrieve them from the db, and then build the transactions based on the session id. As shown in Figure 5.17

To get all transactions from our log, we will select all log entries that contains a buy event. Which is what the following method does:

Listing 5.4: builder/association_rules_calculator.py

```
def retrieve_transactions():
    sql = """
        SELECT *
        FROM Collector_log
        WHERE event = 'buy'
        ORDER BY session_id, content_id
    """
    cursor = data_helper.get_query_cursor(sql)
    data = data_helper.dictfetchall(cursor)

    return data
```

① SQL query to get all buy events.

The `retrieve_transactions` method returns a list of all buy events, this is fine, but we need to group each of them into a transaction, so we will feed the data into the following method `generate_transactions`. This method runs through the data and collects each transaction into a dictionary, containing the transaction id as key and a couple of the ids in the transaction.

Listing 5.5: builder/ association_rules_calculator.py

```
def generate_transactions(data):
    transactions = dict()

    for transaction_item in data:
        transaction_id = transaction_item["session_id"]
        if transaction_id not in transactions:
            transactions[transaction_id] = []
        transactions[transaction_id].append(transaction_item["content_id"])

    return transactions
```

- ① iterate through all rows in the data.
- ② retrieve the session_id (aka session id in our case)
- ③ if not seen before, create a list and add to the dictionary.
- ④ append the content to the transaction.

3: GET ALL ITEM SETS SIZE ONE AND CALCULATE THEIR SUPPORT.

We can now calculate the frequency sets. The following method is just used to make it easier to abstract what is going on. Listing 5.6: builder/ association_rules_calculator.py

```
def calculate_support_confidence(transactions, min_sup=0.01):
    N = len(transactions)
    one_itemsets = calculate_itemsets_one(transactions, min_sup)
    two_itemsets = calculate_itemsets_two(transactions, one_itemsets, min_sup)
    rules = calculate_association_rules(one_itemsets, two_itemsets, N)
    return sorted(rules)
```

- ① N is the number of transactions
- ② calculate all itemsets of size one
- ③ calculate all itemsets of size two.
- ④ now calculate association rules.
- ⑤ sort rules and return.

The method creates two dictionaries one for the frequency sets with just one element and one for the frequency sets with two elements. The declarations are followed by two calls to methods that will populate the dictionaries. Let's look at calculate_itemsets_one first:

Listing 5.7: builder/ association_rules_calculator.py

```
def calculate_itemsets_one(transactions, min_sup=0.01):
    N = len(transactions)                                ①
    temp = defaultdict(int)
    one_itemsets = dict()
    for key, items in transactions.items():
        for item in items:
            inx = frozenset({item})
            temp[inx] += 1                                 ②
    # remove all items that is not supported.
    for key, itemset in temp.items():
        if itemset > min_sup * N:                      ③
            one_itemsets[key] = itemset                ④
    return one_itemsets                                  ⑤
    ⑥
    ⑦
    ⑧
```

- ① N is the number of all transactions.
- ② defaultdict it is dictionary that initializes new elements with default values of the type you use as input.
- ③ Go through each transaction.
- ④ look at each item
- ⑤ FrozenSets are a special type of sets. FrozenSets are immutable and can therefore be used as dictionary keys.²⁷
- ⑥ since we use the defaultdict we don't have to worry about initialization.
- ⑦ go through all elements and
- ⑧ pick the ones that have support larger then the required support.

Defaultdict

In the code above, there is an import of a defaultdict, this is a dictionary where every new element is initialized to the default value of the type of which it was declared. this makes the code a bit more readable. The method runs through all the transactions and for each transaction it increments counters for each element found in the transactions.

When this is done, the itemsets with one element, which were found in the method above, is now fed into the following method which calculates itemsets with confidence and support higher than some minimum.

Listing 5.8: builder/ association_rules_calculator.py

```
def calculate_itemsets_two(transactions, one_itemsets, min_sup=0.01):
```

²⁷ <https://docs.python.org/3/library/stdtypes.html#frozenset>

```

two_itemsets = defaultdict(int)

for key, items in transactions.items():
    items = list(set(items))

    if (len(items) > 2):
        for perm in combinations(items, 2):
            if has_support(perm, one_itemsets):
                two_itemsets[frozenset(perm)] += 1
    elif len(items) == 2:
        if has_support(items, one_itemsets):
            two_itemsets[frozenset(items)] += 1
return two_itemsets

```

- 1 iterate through all transactions
- 2 remove duplications
- 3 look only at the transactions which contain more than 2 items.
- 4 look at all the permutations of two items one can build from the list of items.
- 5 check if the itemset has support
- 6 add the itemset to the list of itemsets.
- 7 otherwise if the transaction only contains two items.
- 8 check if the itemset has support
- 9 add the itemset to the list of itemsets.

The resulting dictionary is iterated once more, and the items that are above the minimal support is added to the output dictionary. Which is eventually returned.

Finally we are ready to calculate the association rules.

Listing 5.9: builder/ association_rules_calculator.py

```

builder/AssociationRulesCalculator.py
def calculate_association_rules(one_itemsets, two_itemsets, N):
    timestamp = datetime.now()

    rules = []
    for source, source_freq in one_itemsets.items():
        for key, group_freq in two_itemsets.items():
            if source.issubset(key):
                target = key.difference(source)
                support = group_freq / N
                confidence = group_freq / source_freq
                rules.append((timestamp, next(iter(source)), next(iter(target)), confidence,
                           support))
    return rules

```

- 1 iterate through all the itemsets of size one.
- 2 for each itemset of size one, iterate through all the itemsets of size two.
- 3 check if the itemsets of size one is a subset of the itemset of size 2.
- 4 if so set target to the element(s) that is not the source.
- 5 support is the number of times the itemset has occurred divided by the total number of transactions.
- 6 while the confidence is the number of times the group occurs compared to how often the source occurs by itself.

It is worth noting that there could be value in also looking at association rules for sets, if you were going to use it to recommend things when looking at the shopping basket, while for us

only calculating it with one source element makes sense because we will use it to show recs in reference to just one content item.

5.5.4 Saving the association rules in the database.

Now that we can calculate the association rules it might be worth considering if this is a good idea to calculate them every time a customer looks at a product.

What we need is to calculate them offline and then a place to save them, where they can be retrieved fast. But the association rules should also be updated, and while the update is going on this should not disrupt the service.

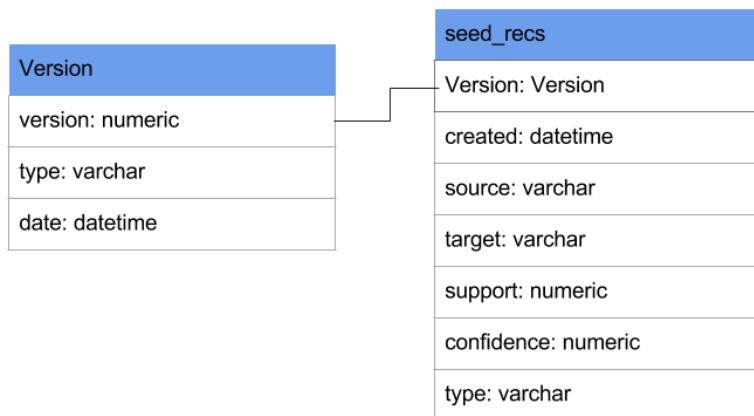


Figure 5.1812 Data model for the association rules

So can we just save the association rules to one table? Well it could give problems. Let's take a step back so we are sure everybody is following. We have users clicking on details pages of movies, which will query a rules table. This will also happen while the system is adding new rules to the system. To avoid having problems while saving new rules, we need a marker that shows which rules are the current ones to retrieve. A way to get around this is to introduce a version table as shown in figure 5.18. The version table will allow the system not to mix up different runs of rules. The versoin table contains a row for each full version of the rules. It means that we can't just query the association rules directly but you would have to join it with the version table like the following:

Listing 5.9: SQL to retrieve association rules from a specific source latest version

```
WITH currentversion as
(SELECT version
   FROM version
  WHERE type = 'association_rules'
 ORDER BY version desc
 LIMIT 1)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/practical-recommender-systems>

Licensed to Asif Qamar <asif@asifqamar.com>

```
SELECT *
FROM seeded_recs recs
WHERE source = '<the source id>'
AND recs.version = currentversion
```

Before you get too excited about the version table however I better tell you that it is not implemented in the MovieGeek website.

5.5.5 Running the association rules calculator.

To run the association rule calculator, you should first be sure that you have generated the log entries which we mentioned in CH 4 (running python populate_logs.py) and then you can run the following:

Listing 5.10: calculate the association rules

```
python -m builder.association_rules_calculator
```

This will generate the association rules and save them in the database.

And with this you should have understood how association rules work and are implemented. Let's have a quick look at the MovieGeek site to see them in action. To retrieve recs using association rules you call the following method, it is very Django specific and maybe not too interesting to look at if you are not into Django. But you should still be able to see what happens.

Listing 5.11: seeded recommendation rules using association rules

```
def get_association_rules_for(request, content_id, take=6):
    data = SeededRecs.objects.filter(source=content_id) \
        .order_by('-confidence') \
        .values('target', 'confidence', 'support')[take:]
    return JsonResponse(dict(data=list(data)), safe=False)
```

- ① retrieve objects from the SeededRecs table where source equals content id and order them by confidence.
- ② wrap in json and return.

Let's have a look at the landing page, shown in the fig 5.19. It now contains the top-10 chart on the right. If you click on one of those in the chart. In theory, you should be able to click on any movie, but remember we built the association rules using only a bit of data, so the risk is that the system simply won't find any association rules.

Movie GEEKs

Genres					
Documentary					1. Teenage Mutant Ninja Turtles
Short					2. The Huntsman: Winter's War
Horror					3. Hail, Caesar!
Adventure					4. Kung Fu Panda 3
Fantasy					5. Now You See Me 2
Action					6. 13 Hours: The Secret Soldiers of Benghazi
Crime					7. Deadpool
Drama					8. Inside Out
Romance					9. The Angry Birds Movie
History					10. Zootopia
Comedy					
Family					
War					
Sport					
Biography					
Mystery					

1 2 3 4 5 »

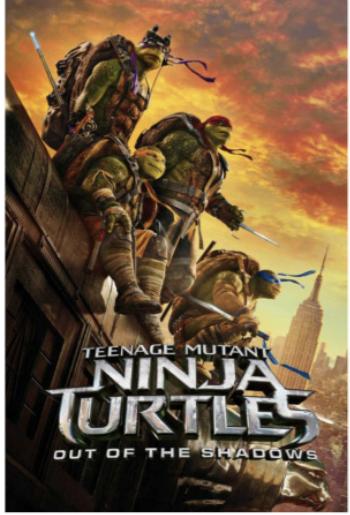
Figure 5.19 landing page with top 10 content on the right.

In my case, I click on *Teenage Mutant Ninja Turtles*, and get the following recs on the details page, as shown in figure 5.21

Genres

- Comedy
- Horror
- Drama
- Fantasy
- Biography
- Thriller
- Film-Noir
- Talk-Show
- Crime
- Western
- Music
- Musical
- War
- Reality-TV
- Sport
- Game-Show
- Short
- History
- News
- Adventure

Teenage Mutant Ninja Turtles: Out of the Shadows



Released:
2016-06-01

Description:
After supervillain Shredder escapes custody, he joins forces with mad scientist Baxter Stockman and two dimwitted henchmen, Bebop and Rocksteady, to unleash a diabolical plan to take over the world. As the Turtles prepare to take on Shredder and his new crew, they find themselves facing an even greater evil with similar intentions: the notorious Krang.

Language:
en

Average rating:
5.8

Genres:
| Adventure | Comedy | A

Buy

Association rules are used to create these recommendations.

Frequently bought with these








Figure 5.20 showing the association rules

5.5.6 Use different events to create the association rules

A friend recommended that I look into a book called “Online Consumer Behavior”, to add some more details on how users behave on the net to beef up chapter 3 (I will leave it to you to guess whether I did it or not?). But looking at the book I came across a new recommendation type on Amazon I haven’t seen before, shown in figure 5.21 think that this recommendation does a disservice to the book, as it gives the impression that people looked at the book, but eventually bought something else.

What other items do customers buy after viewing this item?



Figure 5.21 Amazon.com: Customers who viewed this went on to buy

But it is interesting because it shows a way to beef up your association rules. Because even if not many people bought this book, you could still create association rules, just by finding all the sessions where a customer has viewed the book "Online Consumer Behavior" and then see what was eventually bought. So instead of starting out with frequency sets only containing things bought, you will have all the sessions with the buy events plus the "online consumer behavior" book. Then when you have found all the supported frequency sets, you calculate rules that start with the book and then use those as recommendations.

5.6 Summary

This is so exciting that we don't even need to summarize, and probably you want to jump forward and get started on more ways of constructing recommendations. But summaries are good to refresh what you have been through.

- Charts are great and easy to add. We have talked about how they can be used, and looked at an implementation of one. There are many ways you can build a chart, not only by counting which items has been bought more.
- Never present content without adding some kind of ordering on it. Content should be ordered according to what you think most users are interested in the content. For movies and books that can be production year, for coupons, it can be money saved.
- Association rules are based on what is bought together and used to show frequently bought together type recs. The usefulness of the rules is calculated by looking at
 - Support and
 - Confidence.
- It is good to save recommendations in the database, that will make the recommender system respond faster. But it takes time to calculate them, and they take up space.
- Adding a version to the recommendations in your database enables you to have several versions in the db at once, that means that you can have on which is used in production and then switch to a new one when its ready, but more importantly if something happens with the recs that you are currently using you can revert to an older version.

6

The user (and content) who came in from the cold

Open your arms and put on the big smile it is time to learn how to greet new customers:

- You will understand the cold-start problem, which is related to new customers.
- You'll learn how to segment users such that we can start looking at semi personalization.
- With your newly acquired knowledge we will look at Redbubble.com as a case for cold start problems.
- Finally, you will look at an implementation of a simple personalize recommender using association rules.

We are off to a cold start in this chapter, so put on your hat and gloves and let's get started.

6.1 What is a cold Start?

In the previous chapter, we have talked a lot about how to get data, and luckily most websites will have data before they start adventures into recommender systems. But even a lot of data will not solve the problem of how to introduce new things, that be items or users.

Not so surprisingly, if you don't have any knowledge of the user, you cannot personalize. And no personalization is a huge issue because you really want to make new visitors feel welcome, so they will come back and become loyal returning customers. Repeating customers are good, and we want to keep them happy, but there is nothing like adding a new one to the list.

This problem is so big that it has a name – it is called *cold start*. The cold start problem is a term used not only for the interesting question of how to serve recommendations to new users but also how to introduce new items into the catalog. New items won't show up in any of

the non-personalized recs as it doesn't have the sales numbers to enter into sales statistics, and won't come in personalized recs as the system doesn't know how to relate it to other items.

Under the umbrella of Cold start problems, are also the gray sheep, which are users that have such an individual taste that even if there is data, there are no other users who have bought any of the products the gray sheep has.

Personalized recommendations are built based on of information that binds content together with users. Figure 6.113Figure 6.1 illustrates the most common connections used when we calculate recommendations. In the following chapters, we will get much more into these connections, here I just want to show it to you, to be able to say, that the cold start problem is about figuring out what to do when you have none or petite of this type of connections.

The connections are used to be able to say that if a user has rated film 1 high, then if one of these connections are present we can recommend film 2. If there are no such connections outgoing from the items the current user has rated high, then we will have a hard time recommending something to the user.

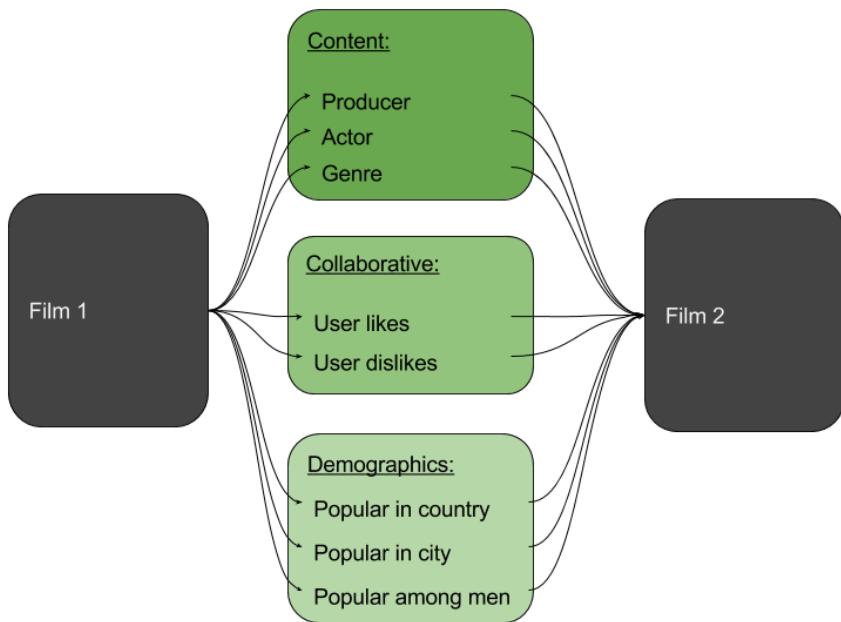


Figure 6.113: What binds content together?

Luckily, this mostly only occurs when we have new users, who haven't related to any items yet (related meaning viewed, rated bought), but as we will see customers with unique taste also gives us the same type of headache.

But I am getting ahead of myself. Let's take it from the top in more detail. Let's start with the easiest one first – cold product – and then move on from there.

6.1.1 Cold product

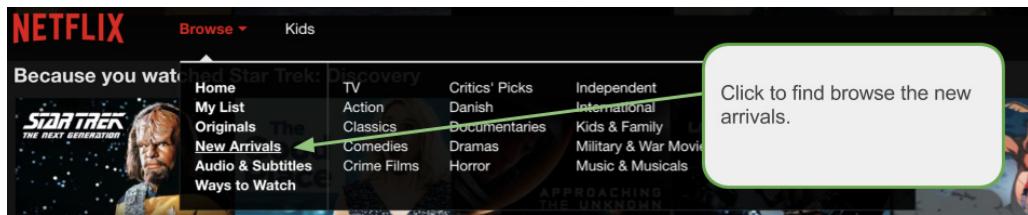


Figure 6.2 Netflix shows new arrivals.

A catalog of items does not need to be very big, before new content will disappear into obscurity, like a needle in a haystack. So, it is crucial that some extra effort is done when something is introduced. In most cases adding new content, should be accompanied by a manual process where the site should do some campaigning on the item, like sending emails to users who likes similar things. An easy way to make visitors notice new content is simply by adding a place that shows new content. Most people love to check out new stuff, on Netflix, it has its own area, on the level of genres like action and comedy, as shown in **Error! Reference source not found.**. Another way is to boost new content such that it looks popular and show up in your recommendations, and then if it doesn't get consumed, slowly let it decay.

6.1.2 A cold visitor

A new visitor that we do not know anything about is also a cold start problem. But for how long? How long do we know too little about a user to start giving her personalized recommendations? Many scientific papers will say that recommendations can't be calculated before a user has rated at least 20 or 50 items. But in real life, the customer expects sites to start delivering recommendations long before that. Many movie sites will ask you to rate five items to get started, but that is not an option with most sites. If a user search for something it's a great way to understand what a user wants, and if the user puts up a search agent, then you are completely sure it is something the user is interested in. Figure 6.3 shows a search I did on LinkedIn. And right there in the upper corner where you are more inclined to see it, is the job alert maker button.

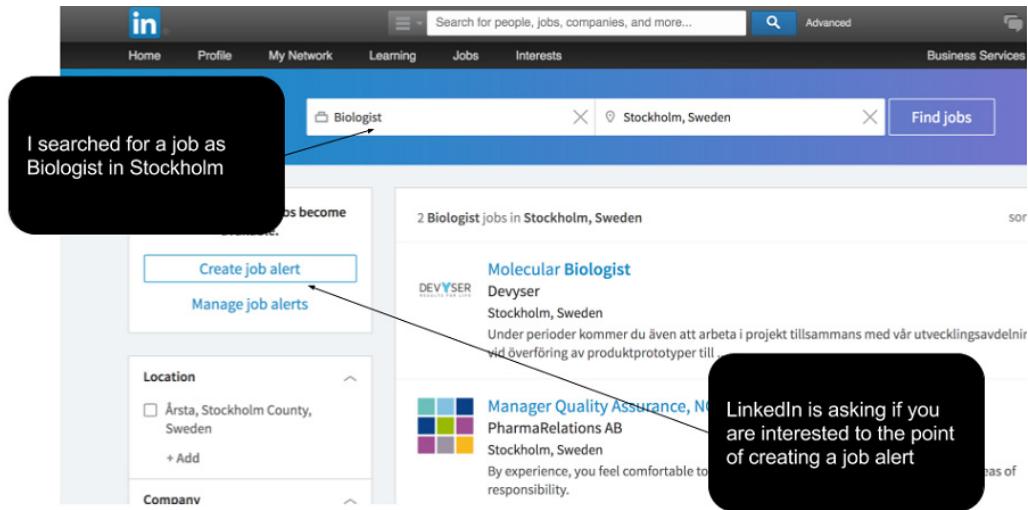


Figure 6.3 LinkedIn is always happy to set up a job alert using your search terms and which tells that the search terms actually did respond to what you were searching for.

When you have enough is a tradeoff, it's a matter of deciding what you want, high quality recommendations that will only arrive when you have data, or lower quality recommendations with less data. Think about it this way: What is the smallest amount of content you will want to have preference data about before you can deduce a taste? In some cases, this could be five, like on a movie site, but in other places it could be different.

When a new user arrives, we know nothing²⁸. No data means no personalization. That is easy, but instead of just throwing the towel into the ring and leaving to drink some herbal tea, let us try to see how much we need to know before we can do something.

One thing to keep in mind is that when you have very little data on a user, you might not have a correct picture of the user's taste.

For example, imagine that Sara arrives at MovieGEEKs and divides her watching like this: Action: 20%, Drama: 20% and Comedy 60%. She knows what she is looking for and goes straight to the drama category and buys one. Now the systems know that Sara likes drama, but not how much, and if she likes only drama. If the recommender assumes that it is only drama from here on, that's likely only a small part of Sara's taste. Therefore, those scientific papers might have something going for them. However, as mentioned before, it's often better to recommend something that is a little off, instead of providing no recommendations at all. In chapter 12 we talk about hybrid recommenders, and a certain type called mixed hybrids, using

²⁸ That is ofcourse a truth with modifications, because we know their IP and therefore probably also where they come from. More on that in section 6.3

such a recommender will simply fall back to most popular recommendations if personalized recommendations are available.

At good old Amazon, they will, among other things, start showing you what people are looking at right now. Finding what people are currently looking at can be done quite quickly. Just look in your log and find what content has been viewed within the last minutes, hour, or day. But there are also other things to do. That's the topic of this chapter.

Even when you do know something about your customers, you can still have a cold start problem, which is called gray sheep.

6.1.3 Gray sheep

A gray sheep is not what you think, (well I have no idea of what you think) but at least not what many would think. It has nothing to do with the sweet, wool-producing animals. Rather a gray sheep is a user that has such an individual taste that even if there is data, there are no other users who have bought any of the products the gray sheep has. The reason for it to be here among the cold start problems is that it creates the same issue of calculating recommendations for users you don't have any data on. Some of the solutions overlap between cold visitors and gray sheep, so it is worth mentioning here as well.

6.1.4 Let's look at some real-life examples

Take a film, let's call it X; nobody ever bothered to even look at it. And here comes a user, who may have just met somebody who was a statist for 2 min in X, which is the only two minutes she ever wants to watch. Now the user has chatted with him online, so she means to get the film to see him in real life. She arrived at exactly this site because this was the only site that had the movie. In this case the user probably wouldn't want any films like this, so recommending her similar films would be a mistake, but we can't know that, and generally, people tend to buy something because they like it. In fact, this user creates more problems than benefits for two reasons, 1) The system will spend a lot of energy looking for similarity, but won't find anything. 2) If the user does buy something more popular, we suddenly have a link between a popular film and something that was only ever viewed or purchased by one user. This might end up making your recommender start recommending the film X to people who like the more popular film. Only after the user has generated a lot of data can we realize that film X is an outlier, and disregard it, this is not a problem when we talk about association rules, then film X will have low support. But when we start talking about collaborative filtering then it can be a problem.

AN EXAMPLE OF GRAY SHEEP AND COLD PRODUCTS

Now gray sheep sound like odd users that you probably cannot recommend anything for anyway, but consider a site like Redbubble.com (see fig 6.4). Redbubble is a fascinating place where artists can showcase and sell art, sold in many different forms from wall art to t-shirts and hoodies. With a content catalog containing many millions of art pieces, and a customer

base from all over the world, they have many customers who have bought something that nobody else has bought.

Art is very hard to categorize, so it is, therefore, hard to say that because somebody likes one piece of art, he will also like this other one. If you combine that with the fact that most items are sold very few times, the result is that it is hard to recommend things even to old customers.

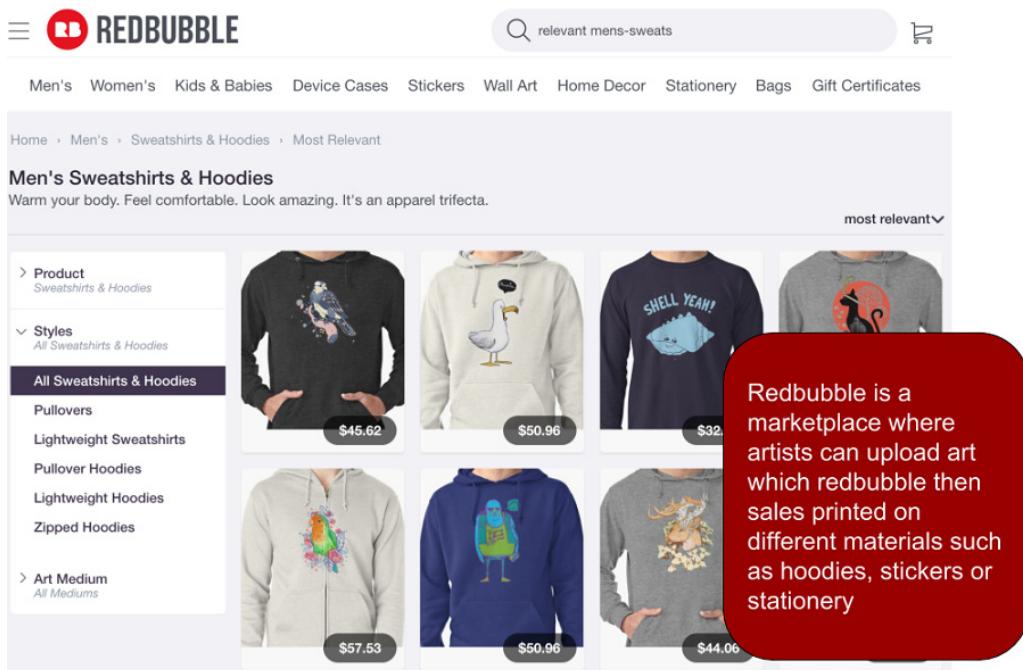


Figure 6.4 landing page of Redbubble

Most of Redbubble's products stay cold products, as very few buy each product, and you, therefore, cannot relate them to other products. And even customers that buy everything from Redbubble can easily be gray sheep, meaning that they have bought things that haven't been purchased by other people.

Redbubble, in many ways, has the same type of problem as news sites like Google News, or video sites like YouTube. Or Issuu, which is YouTube for magazines; artists, moviemakers or magazine makers add content without descriptions like the genre, tags, etc. and when they do add information, people have very different opinions about what a tag means. However, Redbubble is especially interesting, because while sites like Issuu can simply read the uploaded content and try to tag it, it is pretty hard to tag art. Try to have a look around on the site and see if you can make any sense of it.

6.1.5 So, what can we do about cold starts?

In the following section, we will look at different ways to make cold start less problematic²⁹ the cold start problems (user, product, and gray sheep). Many of the solutions for these problems relate to one of the algorithms we will study in the following chapters, so those solutions will be handled then. For example, the cold item is best handled using content-based filtering, which we will look at in chapter 10. I will, therefore, postpone any real attempt to solve this issue until then.

While not actually solving the cold start problem, a solution that often comes up is to offer people to connect using social media accounts like the Facebook Connect, and then extract data from the user's profile, thereby circumventing the cold start problem, it is not said that these profiles will automatically provide you with data that suits your domain, but it could be a start.

6.2 Keeping track of visitors

Sadly, most users are quite elusive. They tend to access websites without signing in first, and from different devices and locations, so it can be quite a feat to recognize returning customers. And this is sad because to be able to understand users we need to know if they are a new user and, even more importantly, we need to understand when they are coming back. We need to track users, new and old, to understand their behavior.

6.2.1 Persisting anonymous users

First of all, as soon as a user arrives it is a good idea to explain the benefits of registering as a user, either through Facebook or manually through a register page. However, while we prefer users to register, we still want to save an id for the anonymous sessions, so we can recognize them if they return.

Recognizing users means that we can accumulate information about the user, which will make the system able to calculate recommendations, even before we know who they are.

In the old days when people would have one stationary computer and no other devices, it was enough just to set a cookie in the user's browser. It is for sure still worth doing it but remember as soon as the user changes browser or device you are lost. This problem is so significant that companies advertise it when they have a solution to it.³⁰

Django enables anonymous sessions, ie sessions where we can place a cookie, but the user hasn't added any user information to the system. The session framework lets you store and retrieve arbitrary data on a per-site-visitor basis. It stores data on the server side and abstracts the sending and receiving of cookies. Cookies contain a session ID. The session ID

²⁹ It said 'get around' here, but as a reviewer rightfully said it is not actually getting around it, it's just making it less problematic.

³⁰ Like Adform for example (see here) link: blog.adform.com/products/cross-device-audience-management

will be looked up in the database, where the data is stored. So, we take care to set a user id and save it so that as long as that cookie is received, we will identify the user, even if it is only on one device. Storing sessions on the server, is something to be careful about, if you have 40 million users then the storage and retrieval of the user data become a problem.

In conclusion, tracking users is hard, even the returning ones, but assuming that you have solved that then here are some ways to get around the cold start problems.

6.3 Three ways to address cold start problem with algorithms

Cold start is still called a problem since nobody has come up with a really great solution. And unlike what some magazines want you to believe machine learning doesn't do magic, it infers things from data. If there is no data with a signal, then there is no reasonable response. So the solutions are about finding information in even more sparse data. We want to use the information we already have in the data, and relate that to a new user, or rather the other way around.

In the following, we will use the association rules we created in the previous chapter, as well as look at making segments of the existing users and then consider how fast we can make users fit into a segment. Finally, have a little chat about how you can ask the user about what they like.

6.3.1 Using association rules to create recs for cold users

Association rules are created looking at shopping transactions and produce rules that tell the recommender that if a user has put bread in their basket, then butter might be a good thing to recommend. In the previous chapter, we used them to make recs on item pages. But what if we stepped back and said that we use these rules in cooperation with what we know about a new user (which is not much), namely the list of items that he has browsed, and then use those as seeds to find relevant rules, and from those rules, do recommendations. Figure 6.4 illustrates how this could be implemented.

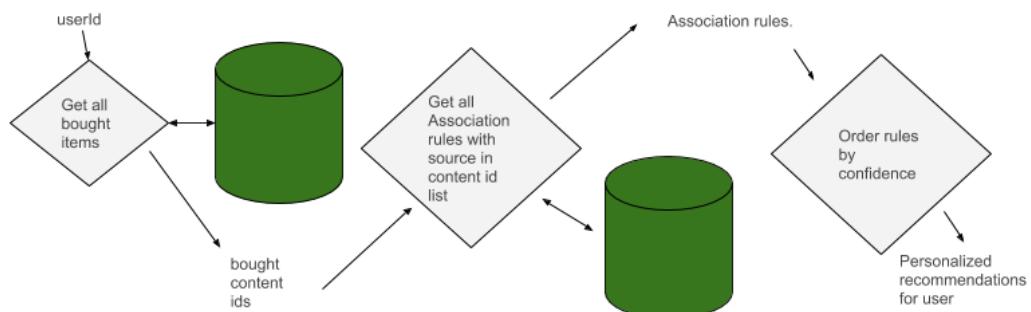


Figure 6.5 Making personalized recommendations with association rules

The system could start when the user does the first item view, which would be what could happen in the first session. Then as one visit became more, the data could be more and more restricted. Something like the following:

Number visits	Events to use
0-2	Item views
2-4	"more details", buys
4+	Buys.

Or you could define it otherwise, and say you start out retrieving only association rules based on the more valuable information. Something like:

1. Get all association rules based on users bought items.
2. Get all association rules based on users bought items + more details views.
3. Get all association rules based on all users data.

It's worth trying out both and understand what will provide the better result.

THE WEIGHTED AVERAGE OF PURCHASED ITEMS.

When a user has bought his first item, we can use the association rules to provide recs based on that item. And as more items are purchased we can take weighted averages of the association rules. So, for example, let's say you bought bread and butter, and the system has saved the following rules:

- bread => marmalade³¹
- bread => butter
- bread => gorgonzola cheese.
- butter => marmalade
- butter => flour.
- Eggs => bacon.

Then we would get that because bread and butter were bought; we have a choice

- bread => marmalade (and butter => marmalade)
- bread => butter
- bread => gorgonzola cheese.
- butter => flour.

But also butter was bought, so we should have added bread => butter rule also, but since butter was already bought, it wouldn't add any value.

³¹ "bread => marmalade" means that there is a pattern in the data that shows that when you find bread in the basket you often find marmalade also.

If two rules point to the same target, we could calculate a new confidence using a weighted average. Or only just order them and take the best ones, and then remove duplicates from the list before returning recommendations.

If the user is a returning customer, but still so new that we have little data, you could also add weights to each of the rules based on recency of when the user has bought the source item. Such that more recent purchases are weighted higher.

A way to make the recs look better, in the movie scenario could be to add some business rules, which means to incorporate domain knowledge into the system.

6.3.2 Using domain knowledge and business rules

Sometimes the recommender cannot do everything for us, for example, figuring out what to show when people buying cartoons also buying horror movies. Eventually, this would end up as an association rules. People who bought Bambi also buying The Texas Chainsaw Massacre would cause the system to offer a chainsaw massacre movie to young people who are just bought their first Disney show!

A way to do that is to filter the content in such a way that we restrict the recommendations to only be of certain types of content which is considered appropriate to recommend based on what type of content the user is currently viewing. Many Data Scientists will tell you that it will spoil the algorithm to add such constraints to the output. But I have found that it is often necessary.

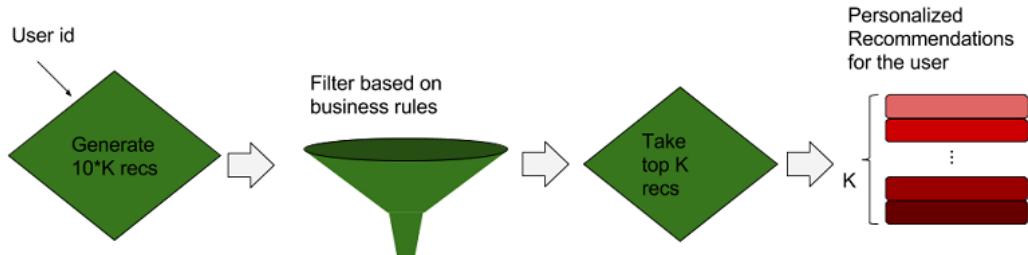


Figure 6.6 Using business rules to provide more sensible recommendations

Business rules can be defined positive or negative. Such that you can say that:

While viewing Cartoons genre, the system can only recommend Cartoons and family films.

Alternatively, you can list all the things that shouldn't appear, like the following:

Never recommend horror from a cartoon.

There are several ways to implement this, but usually, it is done by calculating a list of 100 recommendations (if you need 10) and then do the filtering, and take the top 10 of the remaining, as shown in fig 6.6

6.3.3 Using Segments

Cold start is really an irritating problem, but little can be done about it. But again, if you know something about your users then it might be usable for something?

In Denmark, there is a saying, which says "Similar children play best", I am not sure I have ever understood the expression, as I found that opposites also play well together, but one thing is for sure that "children with similar taste, best watches tv together." Transforming the old proverb would probably make the old people who came up with the original one, turn in their graves, but the transformed proverb is what we are going to use in here. We want to group same taste people, such that we can figure out what kind of people like what kind of content, and when new visitors arrive, and they are of a particular type, then we can just recommend content popular in that segment. This is known as demographic recommendations. Many people say that it doesn't work since people are too different within their demographic groups. But it can get you a bit closer to doing semi personalization. Demographics is often not available without they log in first, but if they do, then you will often have those demographics.

THE OBVIOUS SEGMENTS

So, before we take out the big cannons and start doing clustering, which is unsupervised machine-learning, let us spend a minute thinking about what segments naturally falls out. If you have a visitor you can often get his IP address, with an IP address you are often able to pinpoint where the visitor is calling from. For example it is Copenhagen, and your shop have things that are only sold in a country not Denmark, then you'd probably be better off showing of things that are sold in Copenhagen.

If you sell clothing, then there will usually be at least two groups you want to create: male and female. Beyond that, you can usually also divide users into age groups.

Let's work through an example of how we could take advantage of something like gender. If you have a user database, which contains gender, you can do a simply filter your request when retrieving buy events from users who are female and calculate the chart based on those instead of the whole data. The same, of course, is valid for males. It should be noted that probably there are quite a large percentage of males that are classified as women after buying a dress as a gift (I am one of them) and the other way around.

If we consider a clothes shop and imagine that you would have a chart containing both dresses and suits (the ones that are for males), we would be able to make the men much more at ease by filtering the dresses out, and only serve them men's clothes.

Now, this is a bit silly with clothing because knowing the gender of the visitor is simply a matter of just filtering the content since clothes items usually are tagged with target gender.

But let's move up to level 2 and talk about content, which is not predefined gender specific, we might start seeing the point of this. For example, for different age groups, look at the Star Wars films, which are generally considered boring for people of the generation that were too young to have watched them when they came out and too old to think the next ones were cool. So kids and the generation who watched them in the cinema adores this films. That means that if you knew that people coming into your site were let's say below 15 or between 40-50, then you could recommend Star Wars while the group in between would go for Marvel superheroes movies instead.

THE NOT-SO-OBVIOUS SEGMENTS

Let's try to expand a bit on the obvious segments above and consider the following, what if there were groups like "German women who browse in the evening during the weekend like action" or "American male teenagers buys horror films during school time". Segments like these are not so obvious and might be hard to spot even in the data, but it is valuable information for personalizing the site for the users. Segmentation don't have to include demographics, it can be based on any kind of data you have on your users.

Usually, segments are created by market researchers based on industry practice and wisdom, which in many cases translates to them guessing. Guessing is good, but not always, so instead of doing segments by hand more and more use cluster analysis to find these segments. Cluster analysis is a less subjective way of finding the segments and can be done using unsupervised machine learning. A cluster is a fancy word for a group with similar traits, so we will try to find particular types of users who consume specific content.

6.3.4 Using categories to get around the Gray Sheep problem and how to introduce cold product

Sometimes you need to take a step back to get ahead – the proverb is always very irritating when it is thrown at you, but sometimes this might be a good idea. This is the idea behind the following method of getting around Gray Sheep and some cold products.

If you have a series of products that only a few people have bought/rated it is hard to infer recommendations. But if you take a step out, and use metadata of a product you can maybe use that to find similar products. It sounds confusing, so let's do an example.

Turning back to Redbubble, They have one thing got going for them that artists tend to create (at least I assume) art that falls within the taste of a particular group of users. So a way to get less sparse data is to look at artists instead of art content. To do this, we would need to group all artwork by artist and see it as if it was one content item.

Salvador Dali did many art pieces, two of which are shown in **Error! Reference source not found.** Figure 6.7. And if we imagine that he is not world famous, and thousands would buy his paintings, we could say that instead of trying to look at users who bought the painting to the left also bought X, using the functionality we implemented in the previous chapter.

Instead, we could look at users who bought art by Salvador Dali also bought from artist X. And if it has to go from art to art, then just recommend the most popular of the artist X's art.

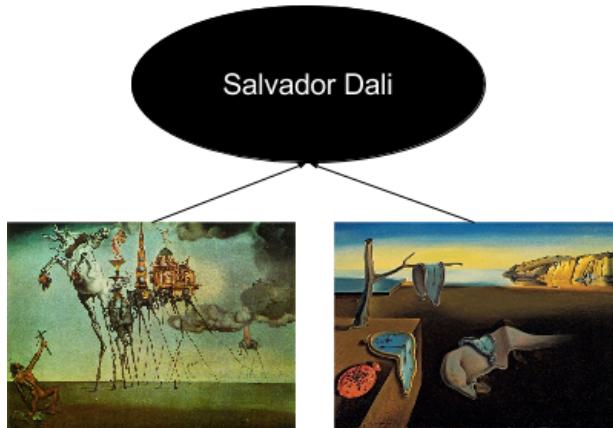


Figure 6.7 Grouping art.

In general, this can be abstracted into the following method, shown in **Error! Reference source not found.** Figure 6.8. This can be used for sites like Redbubble, but also many other sites. With music, you can abstract songs to the artist, with news you can either abstract it to the topic in the shape of tags or the articles author.

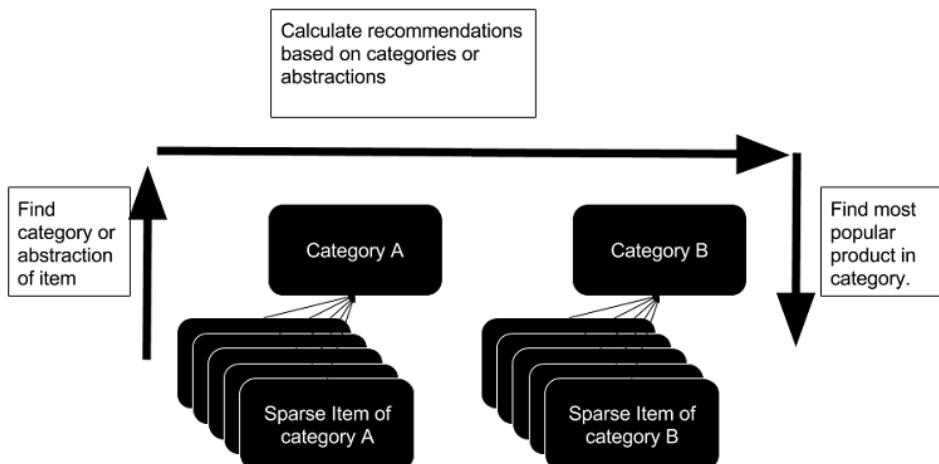


Figure 6.8 using metadata to get around data sparsity

Just remember that the abstraction or classification just can't be too general, as the association between the categories will lose its value. An example of an association not to use is action => comedy. You would be able to find plenty of associations in the data, but will find little value in calculating quality recommendations.

A not so obvious example could be: I like the films of J.J. Abrams, so I would probably go to watch any movie he did (especially after he has done The Force Awakens). So people who like him probably also like Richard Marquand, who directed Return of the Jedi³²If you were to implement this (and we are not going to here), you could create association rules based on the abstraction, and then every time you would ask for a recommendation you could use the current item that the user is viewing. It depends a lot on the dataset that you are holding if this will work.

A classification could also be works of Tolkien or cars that run more than 40 km per liter of gasoline.

To implement this, you could correct the association rules described in chapter 6, but only instead of saving item id, you could save director name of each of them, and then make a special lookup that looks for rules based on the metadata.

6.4 He who does not ask, will not know

One solution to get started is to ask the user what they think about a selected list of items. It will get them started immediately, but on most e-commerce sites it is not a good idea to restrict access until they have answered some questions, so it might not be worth doing it, but make it optional. Amazon does offer you the possibility to improve your recs. This is done by logging into Amazon and click improve my recommendations, as shown in **Error! Reference source not found..**

³² Okay in all honesty the films that Richard Marquand did beyond star wars does not actually seem to be something I would like, but let's continue for the sake of the example.



Figure 6.914 Amazon offers to improve your recommendations

They way Amazon does it, however, does not help us much in this case, as we are trying to collect data about new users not existing ones (even if you can never get too much data.)

So how do we create a page where a user can teach us about their taste. It is not completely as simple as it sounds. Because what content items should you show?

The most popular ones are the ones that everybody likes, but if you like them, just means that you are like everybody else. While showing the least popular ones, will tell a lot about the few strange people who does like them, but most likely, you will spot the gray sheep, and not be any closer to understanding the user. Otherwise, they would not be the most unpopular ones. So the chances are that you would be better off just guessing the recommendations. Another thing to consider is that if you do pick a popular item, you would have the advantage of being able to compare this user to many others.

To solve this problem, we have to turn to something we called active learning, and it is really cool. But sadly, it is beyond the scope of this book to go into. A good place to go and learn more is in the publically available article Active Learning by Ruben et al.³³ To explain active Learning for recommender systems in a few words it's about creating an algorithm, which will come up with good examples for the user to rate, which will provide the recommender with most knowledge about the users taste.

6.4.1 When the visitor is not new any longer

It is worth putting some check if the user is not new any longer; this can do over time, say that we will only use the evidence for the last week for this kind of recs, or the most recent 20 evidence items. Or we might add weights on them, such that new items have a higher weight.

³³ N. Rubens, D. Kaplan, M. Sugiyama. *Recommender Systems Handbook: Active Learning in Recommender Systems* (eds. P.B. Kantor, F. Ricci, L. Rokach, B. Shapira). Springer, 2011. It is part of a book, but you can also find it online for free.

We might also say that when the user has bought five items, we will only use the buy events as seeds.

The association rules seem mostly to make sense when you look at the raw evidence, and not on the implicit ratings, but you could also start using the implicit ratings as seeds, and then weight them based on the ratings.

6.5 Using association rules to start recommending things fast.

So how would we go about adding association rules to the MovieGEEK site? We already have a framework in place for the association rules, so we just need to take what a user has shown an interest in and then find association rules for each of them, and return the ones we think are better. Not a lot of magic here, but let's see if it does the trick. Association rules are just one way this can be implemented, any kind of similarity method could be used. For example, we could use content-based recommendations instead of the association rules, but content-based filtering is in chapter 10, so let's look at association rules for now.

In terms of the MovieGEEK site we will use the space below the other elements on the front page as shown in **Error! Reference source not found.** Figure 6.10.

So the checklist for adding this is as follows:

- Find a good spot on the page
- Collect and use the list of items the user has interacted with.
- Find the association rules
- Order by confidence and show.

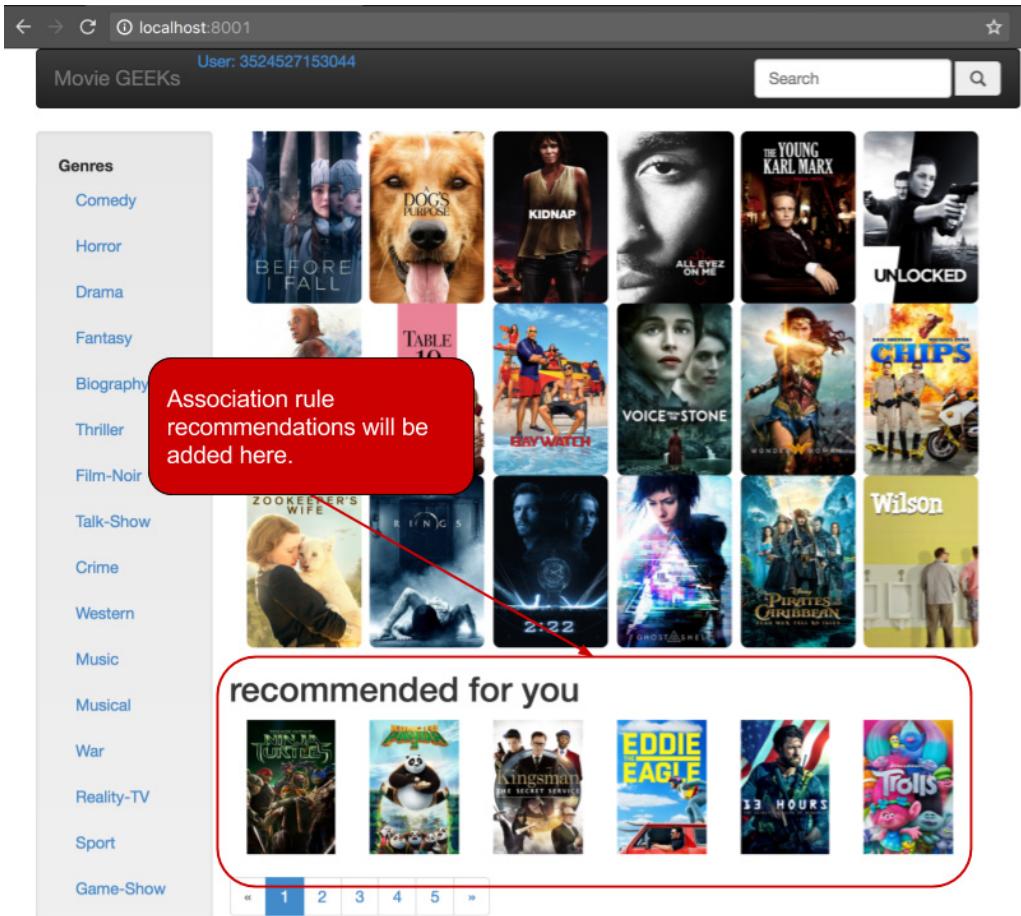


Figure 6.10 Added space for personalized recommendations

6.5.1 Find the Collected items

We want these recommendations to start working as soon as the user has looked at something. So we will get data from the evidence log. Implicit ratings may give a better view of a user's taste, but since we want to service new users, we can't be sure that the system has calculated the implicit ratings yet.

The problem, we are trying to solve here, is to create recommendations for users we don't know much about, so we will use all items the user has interacted with not just the ones she bought. The data is scarce and in most cases the user won't have bought anything yet.

6.5.2 Retrieve Association rules and order them according to confidence.

We could use the method we have already implemented to get association rules, but that would mean that we would query the database repeatedly for each item. Instead, we will create a new method which will query the database by itself.

Listing 6.1: recommender\views.py

```
def recs_using_association_rules(request, user_id, take=6):
    events = Log.objects.filter(user_id=user_id) \
        .order_by('created') \
        .values_list('content_id', flat=True) \
        .distinct() ①

    seeds = set(events[:20]) ②

    rules = SeededRecs.objects.filter(source__in=seeds) \ ③
        .exclude(target__in=seeds) \ ④
        .values('target') \ ⑤
        .annotate(confidence=Avg('confidence')) \ ⑥
        .order_by('-confidence') ⑦

    recs = [{id: '{0:07d}'.format(int(rule['target'])),
             'confidence': rule['confidence']} for rule in rules] ⑧
    return JsonResponse(dict(data=list(recs[:take]))) ⑨
```

- ① Query database for events, related to the user id, order by created, and return a list and make it unique.
- ② Take the newest 20 events.
- ③ Query the association rules and find all the rules where the source is between the content found in the active user's event log.
- ④ And where the targets are not in the user's event log.
- ⑤ Get only the target row.
- ⑥ There might be duplicates targets in the result, so take the average confidence.
- ⑦ Order by average confidence.
- ⑧ create dictionary of the result
- ⑨ jsonify and return.

Calling the method above will produce a JSON output. To test it try requesting <http://moviegeek.com:8000/rec/ar/5/>. It will produce json looking like the following:

Listing 6.2: Output of http://moviegeek.com:8000/rec/ar/5/

```
{
data:
[
{confidence: 0.006463878326996198, ①
id: "1291150"}, {confidence: 0.004617055947854427,
id: "1985949"}, {confidence: 0.004562737642585552,
id: "2267968"}, {confidence: 0.004562737642585552,
id: "0475290"}, ... ②
```

}

- ① note that the values might be different as they are based on a dataset that changes.
- ② the response contained many more items.

6.5.3 Display the recs.

If we create a new private browser tab (on chrome its done by <cltr>-<shift>-n), we will create a new session (see **Error! Reference source not found.**) to play new user. A private browser tab means that you hide all your current cookies (and delete new ones again when you leave the private tab), it is useful in these cases to see how a site looks when you arrive as a new user. Or when you would like to order plane tickets without the companies try to make calculations one which plane you need, and, therefore, make it more expensive. ...Going of trail here... We had a private browser tab, where we can see the MovieGEEKs browser as if it was the first time we visited it.

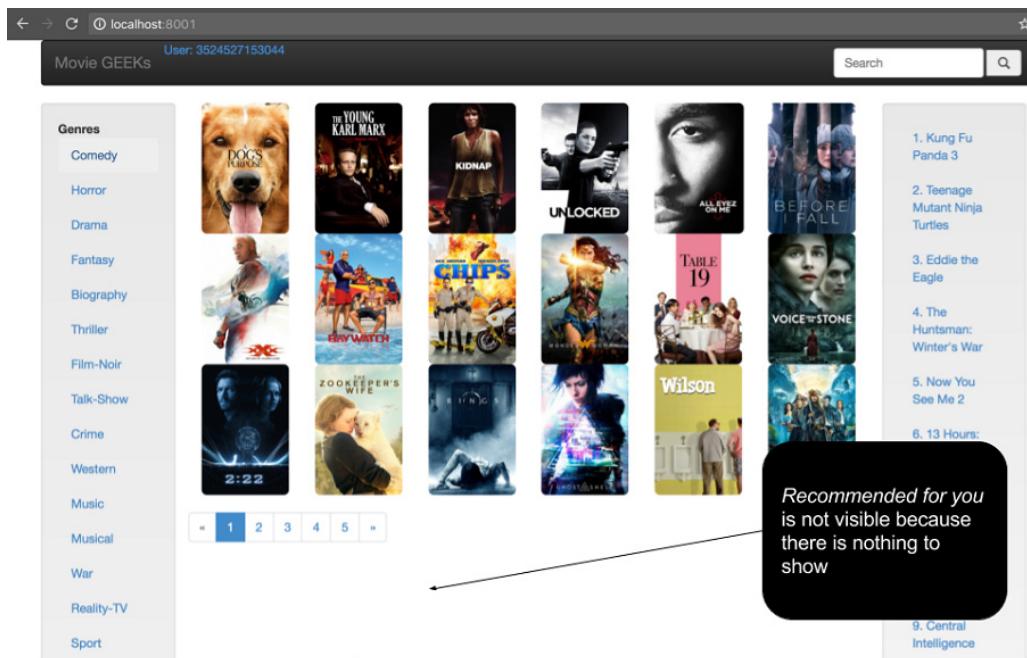


Figure 6.11 new private session, to illustrate what new visitors will see.

Looking at the terminal window where you are running the Django app (see **Error! Reference source not found.**), it is clear why there are no recommendations. Since it is a new private session, the site doesn't recognize the user, and it is therefore considered a new user. Looking further down on **Error! Reference source not found.** it is also shown that it finds no seeds,

meaning no evidence is yet captured. Looking at the association rules above, we can try to buy one of the top content items between the sources.

```
[24/Jun/2016 14:50:03] "POST /collect/log/ HTTP/1.1" 200 2
ensured id: 5103480734263
[24/Jun/2016 14:53:32] "GET / HTTP/1.1" 200 21374
[24/Jun/2016 14:53:32] "GET /static/js/collector.js HTTP/1.1" 200 618
[24/Jun/2016 14:53:32] "GET /rec/chart HTTP/1.1" 301 0
set()
[24/Jun/2016 14:53:32] "GET /rec/ar/5103480734263"
[24/Jun/2016 14:53:32] "GET /rec/chart/ HTTP/1.1"
```

Figure 6.12 ommand prompt is showing new user id being generated.

So let's "buy" a content item (go to <http://localhost:8000/movies/movie/3110958/>), which will lead you to the page you can see in **Error! Reference source not found..**

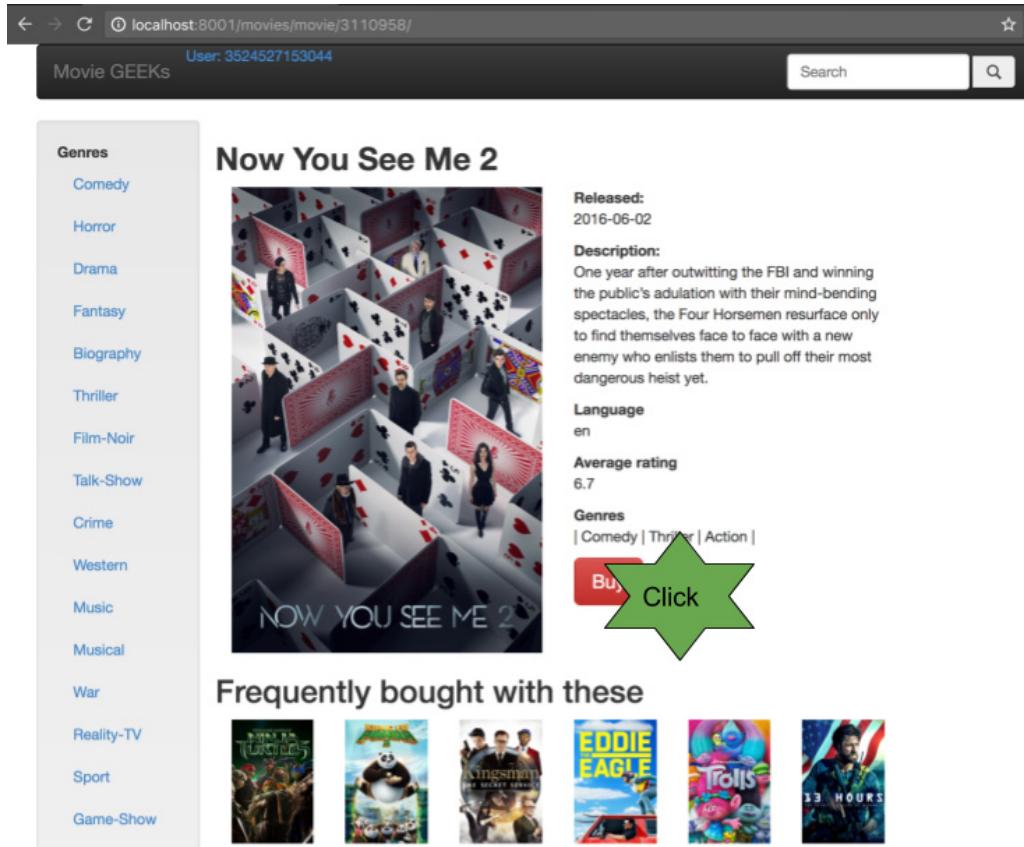


Figure 6.1315 Buying Now You See Me 2.

Click buy and check that your Django app log output shows an evidence item registered.

Now returning to the main page, we should see the same recs as shown in the frequently purchased together seen in **Error! Reference source not found.**, which is also what **Error! Reference source not found.** shows. (it is worth knowing that the recs has content id 10, 18, 45 and 9

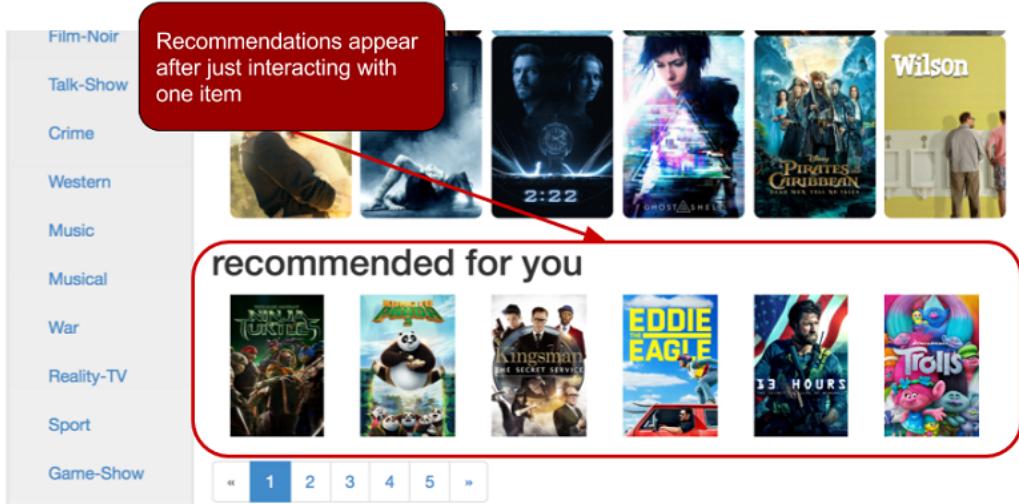


Figure 6.1416 Recommendations after interacting with one item.

Now to make things more interesting let's play we buy another item on the list of association rules above in which case the recs would be updated.

There is a bit of a trick here, if you bought the most popular item, the confidence of the association rules already used will be much larger than others, so before you start throwing something out the window if it doesn't change anything have a look at the confidence first.

If you take a look at the association rules above you can see that if we choose Y then at least on an element should change. So go to the following page: <http://moviegeeks:8000/movies/Y> and click "buy". Then go back to the main window and press refresh (F5 on many most machines and browsers). And the recs should now have been updated.

6.5.4 Implementation evaluation

The recs implemented here are very simple, but the advantage is that it starts giving you recs immediately, which is what we set out to solve. But as we get more data we have better ways to extract taste for it, which we will look at in the following chapters.

Another thing to remember here is that the data we are basing it on is auto-generated, the generation only looks at genres, so an association like "Ninja assign" and "the time-traveler's wife" seems a bit farfetched, think that one is action, and one is drama.

6.6 Summary

We have been around a lot of things in this chapter. Recommender systems comprise of a series of problems, which has a multitude of ways it can be attacked, different studies show

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/practical-recommender-systems>

Licensed to Asif Qamar <asif@asifqamar.com>

one is better than the other, but only for specific data sets, it is, therefore, important to know your options and then try and see what works for your site. To summarize we have looked at the following:

- Cold start problem are the problems of deciding what to recommend to new visitors on a site. It is also considered a cold start problem when introducing new products, we will look at that in more detail in later chapters
- Adding a conscious ordering to the data that is presented is a great start and will make many sites look more dynamic.
- Sometimes Grey sheep's can be helped by abstracting the content into genres.
- Creating recommendations for new users, using association rules, provides an easy and fast way to add personalization that will start working quickly.
- Segments can be build or generated to make semi-personalized recommendations. Alternatively, demographics can be used to create demographic recommendations.

Part 2

Recommender algorithms

An algorithm must be seen to be believed.

– Donald Knuth

In part one you learned about the ecosystem and infrastructure around recommender systems. In part two, we will look at the actual recommender system algorithms. We will look at how to use the data that a system can collect to calculate what things it can recommend to a user. We will also discuss how you can evaluate a recommender system and look at strengths and weaknesses of each algorithm.

7

Finding similarities between users and between content

Similarity can be calculated in many ways, and we will look at many of them:

- You will understand what is similarity and its cousin distance.
- You'll look at how we calculate similarity between sets of items.
- With similarity functions, you will measure how similar two users are, using the ratings they have given to content.
- It sometimes helps to group users, you will do that by using the k-means clustering algorithm.

The previous chapter described non-personalized recommendations and the association rules. Association rules are a way to connect content items together, without looking at the actual item or the users who consumed it. Personalized recommendations, however, almost all contain calculations of similarity. An example of such recommendations could be Netflix's *more like this* recommendation, as shown in figure 7.1. It uses some algorithm to find similar content.

In the subsequent, you will learn several different ways of measuring if two items or two users are similar. To do this we will first look at how to calculate similarity on binary data (bought vs. not bought), and then move on to measuring the similarity between users based on their ratings (and items based on how the same users rate them). With the tools to measure similarity we can use that as an instrument to understand the similarities between the content in the catalog, and in the following chapter, we will put it to good use when implementing Collaborative filtering algorithms.

In this chapter, you will also learn how to cluster users into segments of similar taste. We will use this to provide a way to browse the users. And in the following chapter to optimize the Collaborative filtering algorithms. In other words, it is a bit of a tooling chapter, but it is an essential one, both for recommender systems, but in fact for many machine learning algorithms.

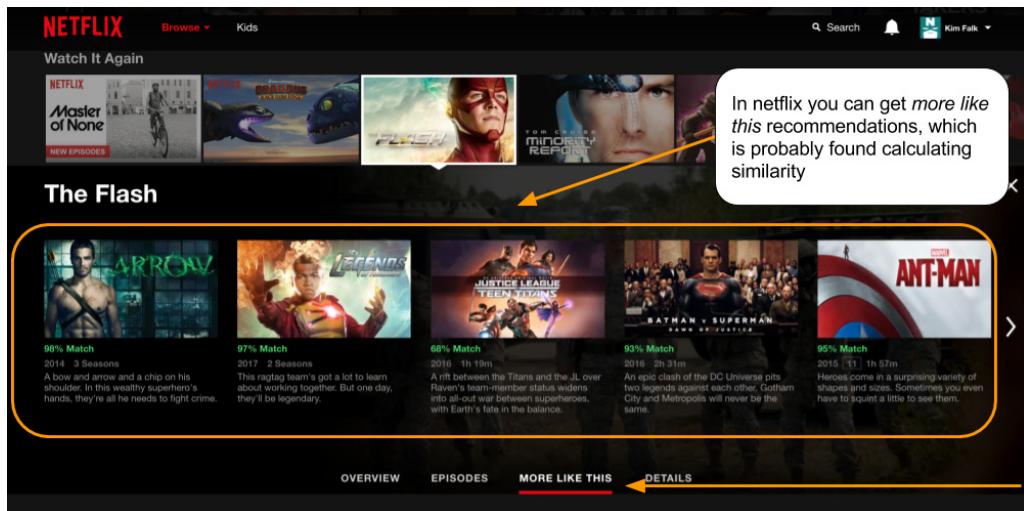


Figure 7.1 More like this recommendation on Netflix based on the Flash series

Let's jump in, but first, let's ponder a bit on the recs in Figure 7.1. It is clear that it is not just about showing things that are similar to *The Flash the series*. If we want attribute a label to *The Flash*, I would say *the first* that comes to mind is "superhero", which would relate it to *the Fantastic Four* movie and the *Justice League* cartoon. But what about the two others? The rest might be similar in other ways. In this chapter we will look at calculating the similarities between content and between users, using different metrics. We will start out with the intuitive explanation of what similarity and then move on to various ways to calculate it.

7.1 Why we need to talk about similarity?

Finding items like the ones you like or users that like what you like.

How do we define similarity? How do you answer the following? On a scale of -1 to 1, how similar are two people? Your first response would probably be to ask *in what sense similar?*

Let's narrow the scope a bit and base the similarity on their taste. Then the answer could also be many things. One could say that two people have similar taste because they both like films with Tom Hanks, science fiction films, or simply all-evening movie³⁴. But then, even people who like sci-fi have different tastes. Maybe one likes Star Trek and the other Star Wars – are they then similar?

Looking at the data we have at hand the possibilities narrows down to use the ratings to try to understand the taste of a user, but similarities could also be found using additional things such as metadata about the content, or demographics of the user. This chapter will look at how we can answer how similar things are. In literature, few similarity functions are said to give good results. And we will go thought each in the following. We will also look at how we can optimize the number of similarity calculations between users, by finding segments of similar users.

7.1.1 What is a Similarity function?

So far, we have been a bit vague about how to define a function that indicates similarity. There are many different ways to calculate it, but the overall problem can be defined as follows:

Given two items i_1 and i_2 , the similarity between them is given by the function $\text{sim}(i_1, i_2)$. This function will increase the more similar the items are. We say that the similarity between the same item is $\text{Sim}(i_1, i_1) = 1$, and two items that have nothing in common will be $\text{Sim}(i_1, i_{\text{nothing in common with } i_1}) = 0$. As can be seen in Figure 7.2 two different similarity functionsFigure 7.2.

Similarity measurement is closely related to the calculation of the distance between items. Distance is therefore also something we will concentrate on in the following.

³⁴ s. In Denmark, we have a term called *heluftensfilm*, which means an all-evening movie, and something that you pay extra to go and see in the cinema, compared to normal length films. A *heluftensfilm* is a film that is longer 2 hours and 45 minutes.

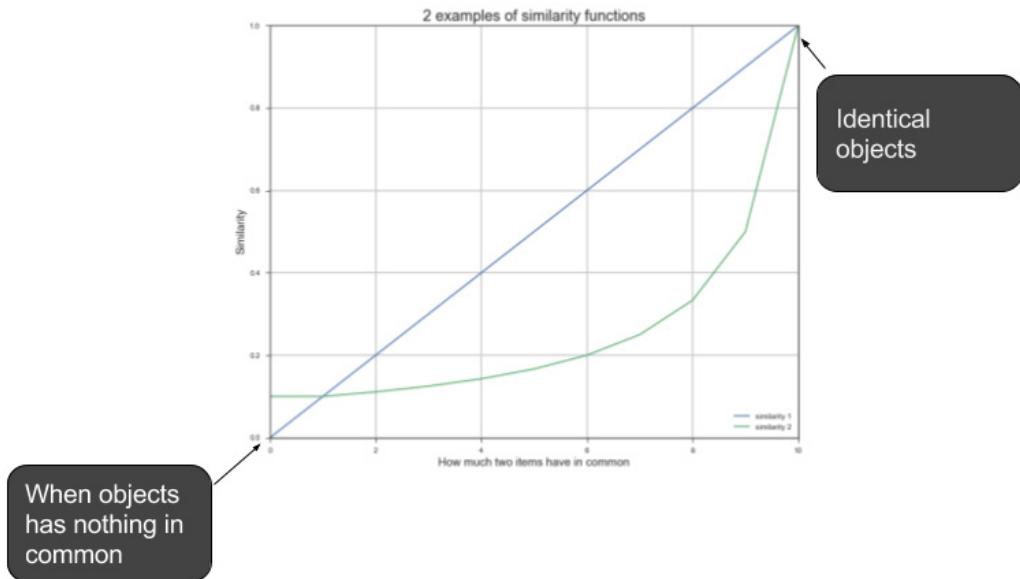


Figure 7.2 two different similarity functions, a straight line will make the similarity 0.5 if it has half the features in common, while the other will only return a similarity of around 0.20.

The relationship between distance is similarity and distance is the following:

When:

- distance gets larger similarity goes toward zero,
- distance goes toward zero then similarity goes towards one.

7.2 Essential similarity functions

As mentioned before, there is no right or wrong similarity method, different methods work better different data sets, but there are some guiding points, which will also be discussed here.

We will start out looking at Jaccard similarity, which is used to compare sets. In our case a set can be the movies a user has bought. We will then move on to look at similarities between ratings, first just one dimension in the form of the similarity between two users rating of one film. This can be generalized to measure how similar users are when they rated many films. To do this we will use Pearson and Cosine similarity functions.

Each similarity method needs a specific kind of data, as shown in the table 7.1:

Table 7.1 Different data types

Data type	Data example	Similarity
Unary data This can be data containing only likes or only transactions of items bought.	User 1 likes movie 2 User 2 likes movie 2 User 3 likes movie 1	Jaccard similarity
Binary data Data where there are two possible values. Like/dislike for example.	User 1 dislikes movie 1 User 1 likes movie 2 User 2 likes movie 2 User 3 likes movie 1	Jaccard similarity
Quantitative Data	User 1 has given 4/10 stars to movie 2 User 2 has given 10/10 stars to movie 2 User 3 has given 1/10 stars to movie 1	Pearson or cosine similarity.

Finally, before getting started, we will just name a few things in table 7.2 that will make it easy to describe the similarity functions below. We define

Table 7.2 Elements of a similarity function

Name	Definition	Example
$r_{U,i}$	Rating of user U for item i	we would write $r_{sara,star\ trek}=4,5$ to indicate that Sara rated Star Trek 4,5
\bar{r}_U	average rating of the user U.	The mean of the rating of all films that has been rated. If Peter rated Star Trek 4, Star wars 3, then $\bar{r}_{Peter}=3,5$
P	set of items rated by both a and b	The set of items, if we have sara and Peter from the examples above. $P_{sara,Peter}=\{\text{Star Trek}\}$ since they both rated that.

7.2.1 Jaccard distance

Originally coined *coefficient de communauté* by Paul Jaccard, who came up with this distance measure to indicate how close two sets are to each other. You will also find it under the name of Jaccard index, or Jaccard similarity coefficient.

So, two sets? What does that have to do with users and content? Well, if we say that each movie is a bag containing all the users who bought the movie. Then we have sets of users, one for each movie. Then we can compare two movies by looking at the two sets of users.

Datasets like the one described above can be produced from the user transactions, aka what user bought a product or not(1=bought, 0=not bought). Or equivalently if the user likes the content item or not, in other words, you have a unary data set, which can be illustrated in a table like the one shown in Table 7.3

Table 7.3 Unary user-item matrix (1=bought, 0=not bought), it is unary because a 1 is information while 0 is no information.



	Comedy	Action	Comedy	Action	Drama	Drama
Sara	1	1	0	0	0	0
Jesper	1	1	1	0	0	0
Therese	1	0	0	0	0	0
Helle	0	1	0	1	0	0
Pietro	0	0	0	0	1	1
Ekaterina	0	0	0	0	1	1

To find the similarity between two items you can see how many users have bought both items, divided by how many users has bought either one of them (or both). Or written more formally it is:

$$\text{similarity}(i, j) = \frac{\# \text{users who bought both items}}{\# \text{users who bought either } i \text{ or } j}$$

If you consider each movie in the table as a bit vector, so *Men in Black* is defined as (1,1,1,0,0,0) while *Star Trek* is equal to (1,1,0,1,0,0). The Jaccard similarity is defined as the number of similar positions in the vector divided by the length of the vector containing all the users who bought either item.

To calculate the similarity between two movies we count the number of equal bits (number of times where the user has done the same with each movie), as shown in Table 7.2. The Table indicates that there are four rows, out of the 6 where the both users has done the same, that means the Jaccard similarity between the two is 4/6. Well almost, because we are not going to look at movies that none of the two had reacted to. So, the actual Jaccard similarity is 2/4 = 0.5. Meaning that they are only a little bit similar, The Jaccard similarity is plotted in Figure 7.2 as similarity 1. Whether 0.5 is a high similarity or not, is very domain specific, so you should play around with the similarity function and see what fits your domain. Later we will look at how users are similar in the MovieGEEKs site using Jaccard.

Table 7.41 Similarity between Men in Black and Star Trek

If you have more details in your data set, you can do some more complicated similarity calculations. As described in the following:

7.2.2 L_p-norms

The L_p-norms are a general way to measure distances, in the section we will look at two different measures, the L₁ and the L₂ norm.

If our dataset is a bit more detailed such as the one we have, with ratings indicating how much people like the content viewed, then we can use a long series of other functions to calculate distances and similarities.

L₁-NORM

What is similarity? I ask again (and hopefully answer). If you want to find if Pietro and Sara have similar opinions about a film like *The Secret Life of Pets*, we could ask them how they would rate it on a scale from 1-10. Pietro thought it was an okay film, so he gave it 6 of 10 rating, while Sara is a dog lover and enjoys cartoons, so it was a nearly perfect film for her, and she gave it an 9 of 10 rating, as illustrated in Figure 7.3 Similarity of two users on one filmFigure 7.3.

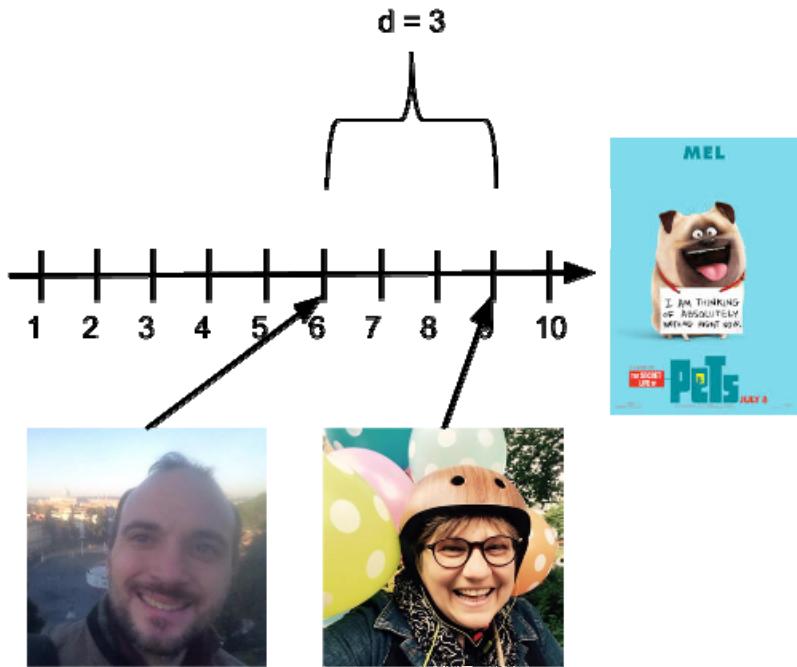


Figure 7.3 Similarity of two users on one film

It is easy to measure how similar two users are based on only one film, even if it might not give any real indication on what their actual tastes are. We can simply calculate the difference between the two. In other words, we have

$$\text{distance}(\text{sara}, \text{pietro}) = |r_{\text{sara}} - r_{\text{pietro}}|$$

Where

r_{sara} is Sara's rating

r_{pietro} is Pietro's rating

With this formula, the distance will be $\text{distance}(\text{Sara}, \text{Pietro}) = 9 - 6 = 3$. Since the maximal distance between the two ratings can be 9, we can do the following to make it a similarity (1 when the distance is minimum, and close to 0 when the distance is maximum). I have added 1 to the denominator, to avoid running into the problem of division-by-zero exception, if the two ratings are the same.

$$\text{similarity}(sara, pietro) = \frac{1}{|r_{sara} - r_{pietro}| + 1}$$

Returning to distance method again and going for two movies, like shown in Figure 7.5. Simply calculate the difference between each rating and sum them, then we will end up with the following formula:

$$\text{Sum of absolute Difference (SAD)} = \sum_{i=1}^n |r_{sara,i} - r_{pietro,i}|$$

This way of measuring distance or similarity goes by the sexy name of Manhattan distance it's part of what is called taxicab geometrics³⁵, in more established circles it simply goes by the name of L₁-norm. The idea is that if you want to measure the distance between two street corners in Manhattan

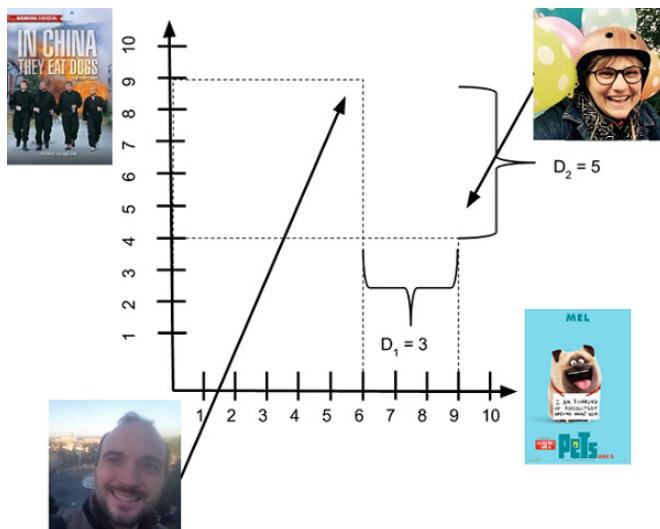


Figure 7.5 Similarity of two users based on ratings of two films, Pietro has rated *In China They Eat Dogs* 9 while Sara only 4 so the difference D₁ = 3, while D₂ = 5 comes from the difference in their ratings for *The Secret Life of Pets*

According to the L₁-norm we would calculate a similarity measure of 3 + 5 = 8.

³⁵ https://en.wikipedia.org/wiki/Taxicab_geometry

Often you will run into the Mean Absolute Error (MAE), which is done using the average of the L_1 norm. As shown in the following formula, only thing new here is that fact that you just divide the sum by the number of items, meaning that you will get the average distance between the ratings.

$$\text{Mean absolute Error (MAE)} = \frac{1}{n} \sum_{i=1}^n |r_{sara,i} - r_{pietro,i}|$$

We will return to MAE in the chapter of Evaluating recommender systems, in scope of similarity we will move on to the next norm.

L₂-NORM

The big brother of L1-norm is the L2-norm, which geometrically can be considered the distance between two points not travelled by taxi in Manhattan but by a bird, going directly from one point to the next. Basically, it falls out of the Pythagoras famous theorem $a^2 + b^2 = c^2$, that states that:

The square of the hypotenuse (the side opposite the right angle) is equal to the sum of the squares of the other two sides.

If you don't know much about Pythagoras, or his theorem, it's no problem, we will move on immediately.

The L2-norm is known as the Euclidian norm and is defined as follows:

$$\text{distance}(Sara, Pietro) = \|r_{sara} - r_{pietro}\|_2 = \sqrt{\sum_{i=1}^n |r_{sara,i} - r_{pietro,i}|^2}$$

When you do machine learning, and you want to, you will often come across the Euclidian norm. It is used to measure how well your algorithm is doing. Again, it is mostly used taking the average of the norm which is called the Mean Squared Error (RMSE), which is also a known fellow.

$$\text{Root Mean Squared Error (RMSE)} = \frac{1}{n} \sqrt{\sum_{i=1}^n |r_{sara,i} - r_{pietro,i}|^2}$$

Again, we can do the trick as we did above by creating the similarity by simply saying one over sum of squared differences. While this can be used it is not something that is regarded as a good solution in recommender systems. They are here since they will be used when we evaluate the algorithms. The following instead is regarded as a good way of measuring similarity.

7.2.3 Cosine similarity

Another way of looking at things is to see the rows of the rating matrix as vectors in space. And look at the angle between them. I know that sounds a bit spacy, but it makes perfect sense once you had a look at the following example.

If we do a tiny example and slim our data down to the following:

Table 7.2 tiny rating matrix



	Comedy	Action
Sara	3	5
Therese	4	1
Helle	2	5

And try to plot it into the coordinate system shown in Figure 7.3Figure 7.5. This of course works for more content items than two as well, but since each item will be another dimension, and no decent ways of illustrating higher dimensions have been found, we will stick with two and hope you believe me when I say it can be expanded quickly to more items.

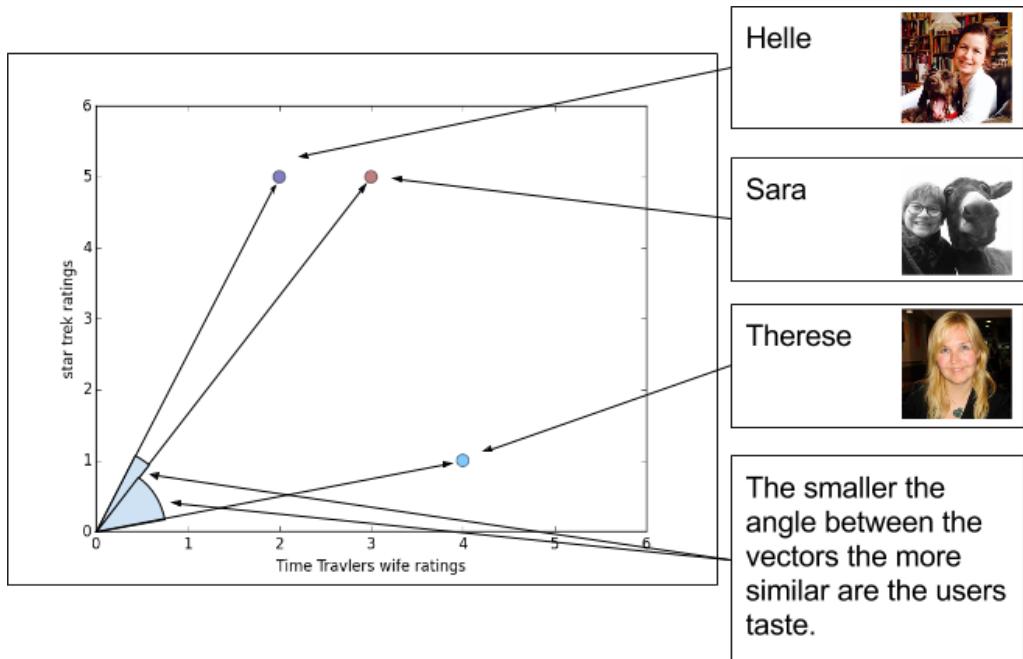


Figure 7.5 showing how you can measure similarity by looking at the angles between the rating vectors.

From the angles between the vectors it is easy to see that Sara and Helle are much closer together than Therese and Sara, so we will take that to assume that the taste of Helle and Sara or more similar. There is a small issue here that always makes you stare at things for long, which is the fact that if we look at the figure 7.6 and imagine that there was a user who had rated both 6 and another one that had rated both 1 their vectors would point in the same direction and therefore they would have similarity one. This is a problem calculating similarity using the angels between their vectors, in practice it is not a problem.

Let's take it for a test run, by computing some similarities. Instead of users, we will look at item similarity. Amazon has shown us that Item-item collaborative filtering is much more efficient.

COMPUTE ITEM SIMILARITY

Looking at item similarity means that we look for similarity between the columns, instead of rows as above:

Table 7.63: Rating matrix


	Comedy	Action	Comedy	Action	Drama	Drama.
Sara	5	5		2	2	2
Jesper	4	5	4		3	3
Therese	5	3	5	2	1	1
Helle	3		3	5	1	1
Pietro	3	3	3	2	4	5
Ekaterina	2	3	2	3	5	5

The function to calculate the angle is done using the Cosine formula which most kids learn in school. In case you forgot what, you learned in school, I will just add the formula here again:

$$sim(i, j) = \frac{r_i \cdot r_j}{\|r_i\|_2 \|r_j\|_2} = \frac{\sum_u r_{i,u} r_{j,u}}{\sqrt{\sum_u r_{i,u}^2} \sqrt{\sum_u r_{j,u}^2}}$$

It is beautiful isn't it...

Sadly, we have to change the function a bit, because since we are talking about comparing different user's ratings and users use difference rating scale when they are rating (a happy rater rates higher than a sad one), we need to take that into account also. For this purpose, Sarwar and friends³⁶ came up with an adjusted cosine similarity to offset this drawback by subtracting the users average rating. Luckily, that doesn't make the formula any less beautiful:

³⁶ Sarwar et al. "Item-based Collaborative Filtering Recommendation Algorithms"

$$sim(i,j) = \frac{\sum_u(r_{i,u} - \bar{r}_u)(r_{j,u} - \bar{r}_u)}{\sqrt{\sum_u(r_{i,u} - \bar{r}_u)^2} \sqrt{\sum_u(r_{j,u} - \bar{r}_u)^2}}$$

7.2.4 Pearson Similarity

FINDING SIMILARITY WITH PEARSON CORRELATION COEFFICIENT

If you take the ratings we looked at above, we have an idea of who is similar, but if you plot them into a diagram with the items along the x-axis and the ratings along the y-axis and then draw a line from point to point, it is even more visible which are similar and which are not. Pearson correlation coefficient looks at these points and measures how different each point is on average. Very different means that it returns something close -1 while very similar means something close to 1. Note that this will answer from -1 to 1 not 0 to 1 as we saw above.

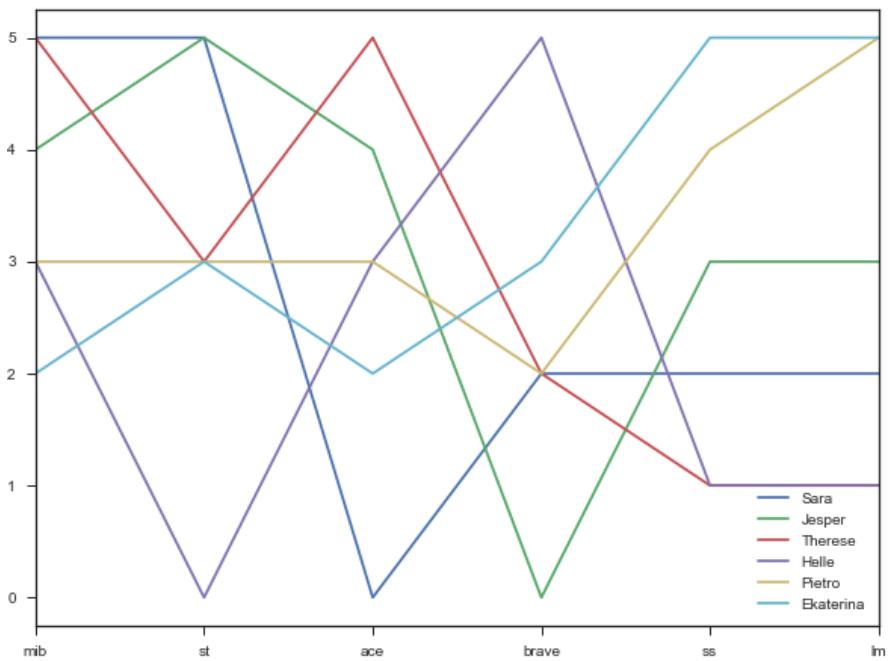


Figure 7.6 charting the ratings of some users. The vertical axis shows ratings, and the horizontal axis is a list of films.

Basically, what happens is that it will calculate how much the two lines drawn between the points of each user respectively, if their trends are identical i.e. it goes up and down the same time, it will be 1 or close to that. 0 means that there is no connection at all, while -1 means that what one user likes the other one dislikes.

The Pearson correlation is calculated by subtracting the users average rating from his using the following equation:

$$sim(i,j) = \frac{\sum_{u \in P} (r_{i,u} - \bar{r}_i)(r_{j,u} - \bar{r}_j)}{\sqrt{\sum_{u \in P} (r_{i,u} - \bar{r}_i)^2} \sqrt{\sum_{u \in P} (r_{j,u} - \bar{r}_j)^2}}$$

where P is the set of users which have rated both i and j .

Do not worry if it looks a bit scary, it is not really, it is just addition and multiplication, as you will see when we implement it later in the chapter.

Let's take it for a test run and see how similar Jesper and Pietro are in taste.

7.2.5 Test running Pearson Similarity

We got the following data on Jesper and Pietro:



	Comedy	Action	Comedy	Action	Drama	Drama.
Jesper	4	5	4		3	3
Pietro	3	3	3	2	4	5

To calculate the Pearson similarity, we need to go through the following steps:

1. Calculating the averages ratings:
2. Normalizing the ratings
3. Put the results into the formula

Calculating the averages ratings:

First, we need to calculate the average rating by adding them all together and dividing by the number of ratings:

$$\text{Jesper: } (4+3+4+4)/4 = 3.75$$

$$\text{Pietro: } (3+3+2+5+4)/5 = 3.4$$

Notice that even if Jesper and Pietro only have four ratings in common, we will still use all of Pietro's five ratings to calculate the average. We are deducting the average to make the ratings comparable, if Jesper only uses the ratings 3 and 4, then it would mean that things he rated 3 were something he didn't like much and four something he liked a lot. While if he gave

ten other movies the rating of one, his average would be much lower, and that would push the value of a rating of four to much more positive. It could also indicate that Jesper only seen films he is somewhat indifferent to, but for this to work we assume that he has used only ratings only 3 and 4 to describe his taste.

NOTE ON IMPLEMENTATION:

REMEMBER when you have an array of ratings, like the one of jespers, you need to do the mean of the items that was rated. If we look at the ratings of jesper we would have the following array [4, 3, 0, 4, 4], if you use a normal mean operation on that you will have 15/5, but he only did four ratings so we don't want to include the zero.

Normalizing the ratings:

Normalizing means a broad range of things, but in its simplest form, it means that you adjust the scale of some variables to be comparable, or on the same level. As mentioned above a certain rating of a user, should always be viewed in relationship to other ratings of the same user. So to compare ratings of two users we need to normalize them so we can compare. This can be done by subtracting the mean for each of the users to their ratings.

Let's subtract it from the ratings of each user (calculating each of the \bar{r}_a and \bar{r}_b).

By doing that, you will have the essential building blocks for calculating the similarity according to Pearson.

Jesper	0.25	-0.75	0.25	0.25	
Pietro	-0.4	-0.4	-1.4	1.6	0.6

Normalizing the ratings around zero means that the ratings get a positive and negative feel to them. A rating of -0.75 sounds quite negative compared to a rating of 3. It probably should be viewed as negative as such, but for an optimistic guy, the scale goes from likes a little to likes a lot while another could go from HATE to like a little.

Put the results into the formula:

Let's set $nr_{a,i} = r_{a,i} - \bar{r}_a$

Then we can transform the Pearson correlation into the following which is the same but uses the normalized ratings instead.

$$\text{sim}(a, b) = \frac{\sum_{i \in P} (nr_{a,i})(nr_{b,i})}{\sqrt{\sum_{i \in P} (nr_{a,i})^2} \sqrt{\sum_{i \in P} (nr_{b,i})^2}}$$

Inserting the normalized ratings we get the following:

$$\text{sim(jesper, pietro)} = \frac{(0.25)(-0.4) + (-0.75)(-0.4) + (0.25)(-1.4) + (0.25)(1.6)}{\sqrt{(0.25)^2 + (-0.75)^2 + (0.25)^2 + (0.25)^2} \sqrt{(-0.4)^2 + (-0.4)^2 + (-1.4)^2 + (-1.6)^2}}$$

Now it's just a matter of calculating (or rather copy paste into Chrome to do the calculation for you)

$$\text{sim(jesper, pietro)} = \frac{0.25}{\sqrt{0.75 * \sqrt{4.84}}} = 0.13$$

This result shows what we could see from the beginning that their tastes are not similar at all.

7.2.6 Pearson is really similar to cosine

Pearson and Cosine similarity looks quite a lot the same, and in fact further down we will see that the cosine similarity function will be extended into the adjusted cosine similarity function. Adjusted means that we add normalization of ratings, and then they are in fact the same function as the Pearson.

With the only difference being that the Pearson function will only use the items that BOTH users have rated while the cosine similarity function will use all rated items by either or both, setting the ratings to zero when one of the users haven't rated it.

7.3 K-means clustering

As we will see in the next chapter, calculating the similarity between users or items is the Achilles heel of the neighborhood collaborative filtering algorithms we will look at in chapter 8, since it requires that the algorithm has an opinion about how similar the currently active users is to all other users in the system. It is therefore, a good idea to divide the dataset into smaller groups, so we only need to calculate similarity in those groups.

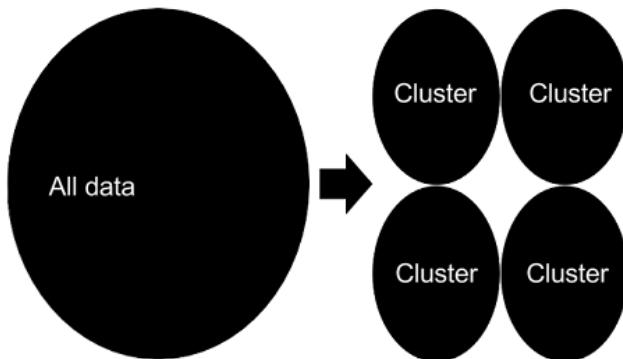


Figure 7.7 Clustering data

Figure 7.7 shows the data clustered into 4 clusters. Consider that every time a user comes to your site you potentially need to compare all the users in your system with the active user. While if you have clusters, you can do with just looking up which the group the user resides in, and PRESTO you have cut away $\frac{3}{4}$ of the users.

7.3.1 k-means clustering algorithm

Clustering is also called segmentation. We talked about it in an earlier chapter, when we talked about non-personalized recommendations. Back then we talked about it in regards to use it to find groups of users that are similar to a new visitor, based on demographic data. The purpose of here is more an optimization since we want to find clusters of users, to narrow down how many times we calculate user-user similarity. If user X comes to our site, we would potentially have to iterate through all the users in the database, while if we can divide the users into clusters we can just look up which cluster the user is part of and do similarities between those. There is, of course, a risk that the group doesn't contain the most similar users, but that is the general idea. Let's try to do some clustering. We will use k-means clustering, which is the most used one.

HOW DOES THE K-MEANS CLUSTERING ALGORITHM WORK?

K-means clustering is what is called unsupervised machine learning algorithm. It is unsupervised because we don't give it any examples of what correct input-output pairs could look like. It is also a parametric algorithm, because we need to give it a parameter k for it to run. Adding just one parameter sounds simple, but it's hard to get the right one, and sadly the difference can be that you don't get nice groups of users.

The parameter k is used to tell the algorithm how many clusters it should find. In the following, you will learn how k-means works using a simple python implementation and we will solve the following silly problem;

Divide the users Sara, Dea, Peter, Mela, Vlad, Pietro and Kim into two groups based on how much they liked two films, the data is as follows:

Small dataset used for k-means

```
Sara = [7,1]
Dea = [10,0]
Peter = [0, 6]
Mela = [1, 4]
Kim = [5,3]
Helle = [9,9]
Egle = [2,1]
Vlad = [4,4]
Jimmie = [6,8]
```

It should be pretty clear how to group the first four as they like one film and not the other, while the rest can be a bit difficult to place. Figure 7.8 shows how they look in a plot.

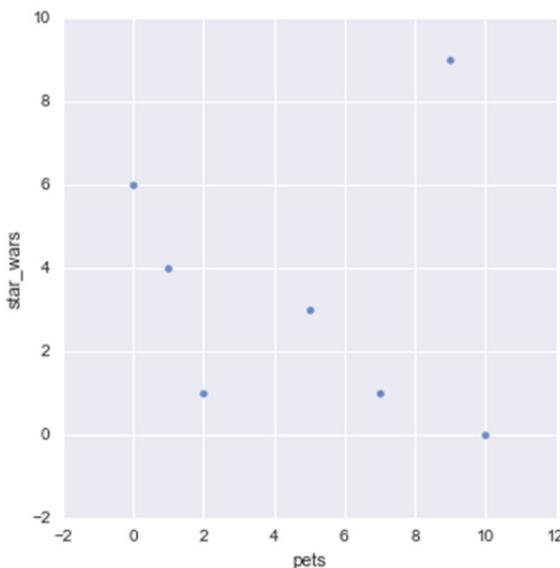


Figure 7.8 each dot represents a user's rating of Star Wars and a film called Pets.

The k-means clustering algorithm works by finding k points called centroids, which satisfy that the sum of the distances between all items and their assigned centroids are as small as possible. Writing this I am sitting watching one of those sprinklers watering flowers placed around, it takes a lot of consideration where to place the sprinkler, as the water here is precious, so we need to find k spots where it will water everything, using as little pressure as possible. The k-means clustering can be used for that too because it produces centroids that would be the place to add the sprinkler.

The algorithm goes through the following steps:

- Select k places, as the centers of the clusters.
- Loop over the following:
 - For each data point in the dataset find the centroid with the shortest distance.
 - When all points are assigned, calculate the sum of all the distances between the item and its centroid.
 - If the distance is not smaller than the previous run
 - return the clusters.
 - Move each centroid into the center of the assigned cluster.

There are many ways to choose these initial prototypes, and it is an important step, because it can change the outcome of the clustering. Please have a look at chapter 10 in *Machine Learning in Action*³⁷, where you also find ways to make the clustering algorithm smarter. What I want to teach here is an overview of how it works, because having a sense of how these algorithms work, will enable you to understand the output much better, and also have a sense of what is right and wrong.

7.3.2 Translating k-means clustering into Python

The code in this section is used for you to understand the algorithm better. The code won't be used in the MovieGEEK app, since it is a nice illustrative implementation, but not very fast. If you want to play around with the code yourself. Have a look at the Jupyter notebook which can be found in the notebook folder in the Github project.³⁸

So, to start our journey into the k-means clustering, we will take the easiest of ways, selecting at random between the input items that will serve as the initial cluster centers.

Randomly selecting items to be centroids Listing 7.1: Generate centroids

```
import random

def generate_centroids(k, data):
    return random.sample(data, k) ①
```

① simply takes out k random elements of the data

FOR EACH DATA POINT IN THE DATASET FIND THE CENTROID WITH THE SHORTEST DISTANCE.

Next thing to do is to calculate the distance between the items in the data and the centroids and find which is closer.

The distance we use here is a well-known fellow: the sum of squared difference, as described earlier in this chapter, the formula was the following:

$$\|r_{sara} - r_{pietro}\|_2 = \sqrt{\sum_{i=1}^n |r_{sara,i} - r_{pietro,i}|^2}$$

This formula can be implemented in the following way:

Listing 7.2: Calculating distance between two vectors

```
import math
```

³⁷ Peter Harrington's machine learning in action

³⁸ <https://github.com/practical-recommender-systems/moviegeek/blob/master/notebooks/ch%207%20k-means%20clustering%20of%20users.ipynb>

```
def distance(x,y):
    dist = 0
    for i in range(len(x)):
        dist += math.pow((x[i] - y[i]), 2)
    return math.sqrt(dist)
```

1
2
3

- 1 iterate through each dimension
- 2 add the squared difference between the vector in that particular dimension.
- 3 return the square root of the sum.

The distance method can be used to decide which cluster each element should be part of, as shown in the following method. The method will return the centroid with the smallest distance to the item (the most similar one)

Listing 7.3: Assign data item to cluster

```
def add_to_cluster(item, centroids):
    return item, min(range(len(centroids)),
                     key= lambda i: distance(item, centroids[i]))
```

- 1 runs through each cluster centre and returns the one that is closer to the item.

When all points are assigned, the distance method is used to calculate the sum of all the distances between the item and its centroid. The sum is used to compare the iterations of the algorithm. This means that the second iteration will compare the sum of the distances in the second setting with the ones before, and if the previous one was better than the algorithm stops otherwise it does another iteration.

For each iteration, the centroids are moved. This can be done again in many ways, in the following method the centroids are moved to the centre of all the points in its cluster.

Listing 7.4:Moving centroids

```
from functools import reduce

def move_centroids(k, kim):
    centroids = []
    for cen in range(k):

        members = [i[0] for i in kim if i[1] == cen]
        if members:
            centroids.append([i/len(members) for i in reduce(add_vector, members)])
```

1
2
3
4

- 1 loop through k clusters, to create new centroids.
- 2 Find all members of this cluster (remember each item was a tuple of two elements the actual vector and the cluster assignment).
- 3 If the cluster is not empty
- 4 add all vectors and divide it by the number of vectors. This means that we will get a center which is in the center of the centroid.

The add_vector is quite simple, it simply iterates through the vectors and add each element.

Listing 7.5: Utility method

```
def add_vector(i, j):
    return [i[k] + j[k] for k in range(len(j))]
```

Now let's look at the whole method

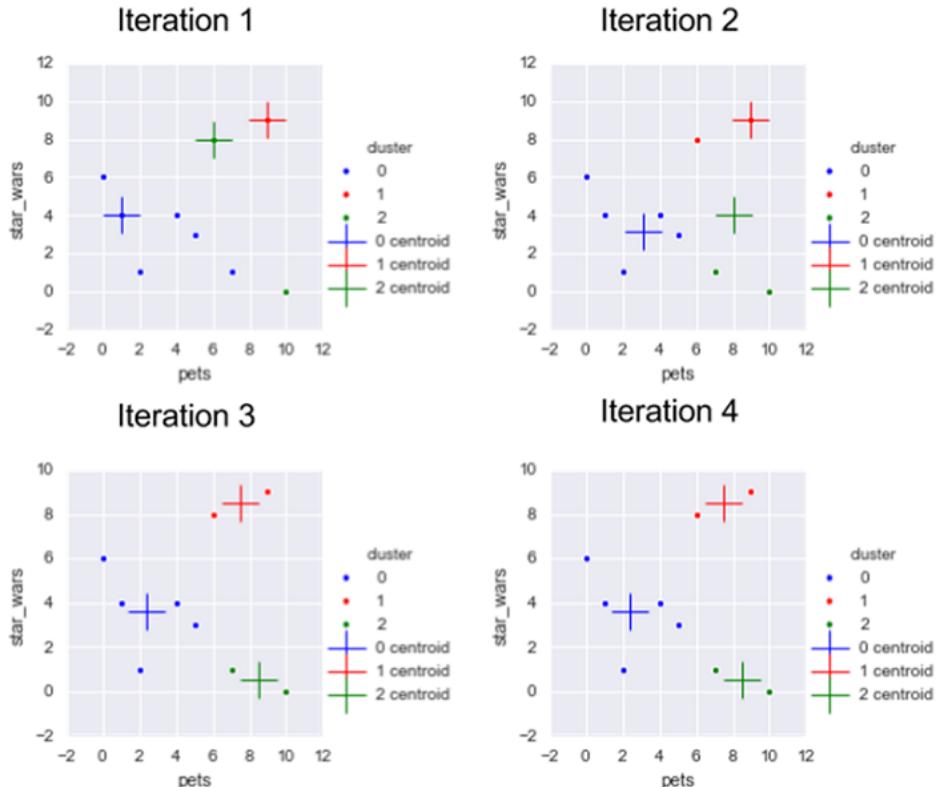


Figure 7.9 the four iterations the k-means clustering went through before it settled.

Listing 7.6: The k means clustering algorithm

```
def k_means(k, data):
    best_weight = math.inf
    ①
    centroids = generate_centroids(k, data)
    ②
    while True:
        iteration = list([add_to_cluster(item, centroids) for item in data])
        ③
        new_weight = 0
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/practical-recommender-systems>

Licensed to Asif Qamar <asif@asifqamar.com>

```

for i in iteration:
    new_weight += distance(i[0], centroids[i[1]])
    ④

    if new_weight < best_weight:
        best_weight = new_weight
        new_weight = 0
    ⑤

    else:
        return iteration

    centroids = move_centroids(k, iteration)
    ⑥

k_means(k, data)
    ⑦

① best weight so far is infinity
② generate the centroids
③ appoint each point to a cluster
④ calculate the distance between each item, and its centroid
⑤ if the new weight is better than the best weight, we continue otherwise we return.
⑥ recalculate centroids
⑦ run the clustering.

```

Now having the clusters means that we can reduce the number of users we need to compare when calculating similarity to the other members of the clusters.

NOTE OF WARNING:

Before moving on, I think it only fair to tell you that when you see an example like the one above it is easy to believe that k-means clustering is some magic beast which will respond to your every bidding. But I must warn you that it is not so. K-means clustering as most other machine learning algorithms is hard to make work correctly. Often they respond with results, which are not understandable nor usable. Examples in books are simple and are constructed to teach how it works, but sadly how it doesn't work is much harder to describe, and is usually left to the reader to discover. Figure 7.10 shows a list of other results I got when I first started out with different starting points.

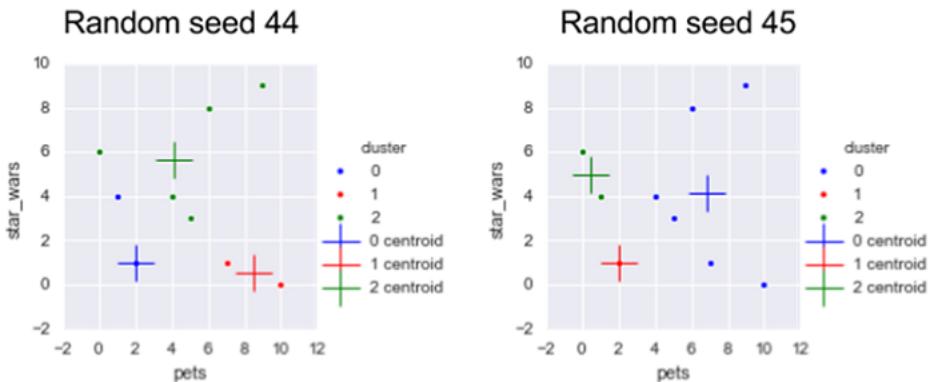


Figure 7.10 some results of k-means clustering that doesn't look very good

And I am afraid I am not going to offer much more assistance. In the next section, you will see how k-means are implemented in the MovieGeek site, to get quick lists of similar users. At least that will show you a more realistic example of how to implement it.

HOW TO USE THE CLUSTERS

There are several ways we can use the clusters.

Look up clusters for existing users

So, if we need to find similar users to say kim, then we just have to look up which cluster he is in, and using that find the other users in the cluster.

Place new users in clusters

But what if a new user arrives. Well then, it's just a matter of finding the centroid which is closer to the new data point and you can use that to find which cluster to look in.

It's tempting to implement clusters now, that we have it fresh, but let's follow the flow of the chapter and go back to the start again and add similarities to the MovieGEEKs site, and then clusters afterward.

There are many different ways cluster users, an alternative way of doing it could also be to use Principle Component Analysis (PCA)³⁹

³⁹ <http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html>

7.4 Implementing Similarities

If you think about it, it only makes sense to look at users that have rated many of the same items as the current user. And to be even pickier we need users that not only rated many of the same items but also rated them the same as the active user rated, which is the ones which we will use to recommend.

If we looked at sets, it would look like what is drawn in Figure 7.11 showing which set of items are important for user collaborative filtering.

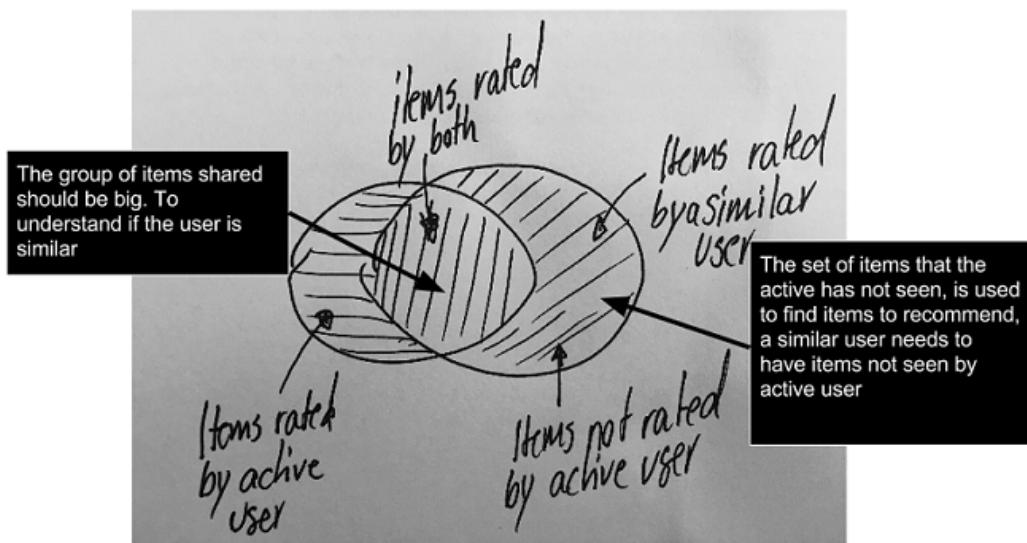


Figure 7.11 showing which set of items are important for user collaborative filtering

So, in short, we want to find users, who have a lot in common with the active user, but not everything. How can we do that? Well if we were doing SQL I would start out doing something like the following:

Listing 7.7: Getting candidate users using SQL:

```
WITH au_items AS (
    ①
    SELECT
        distinct(movieid), rating
    FROM
        public.ratings
    WHERE
        userid = '4')

    ②
    SELECT userid, count(movieid) overlapping
    FROM public.ratings
    WHERE movieid IN (SELECT movieid from au_items) and

    ③
```

```

userid <> '4'
AND overlapping > min
group by userid
order by overlapping desc;

```

4
5
6
7

- 1 start out getting all the items that the current active user has rated.
- 2 now find all users who has rated one or more of the same items the active users has rated.
- 3 Only find ratings on movies that the active user has rated.
- 4 users that is not the active user.
- 5 require the overlap to be more than some min.
- 6 then group by id
- 7 and order by overlapping item.

To make this in Django we can use the QuerySet and our method will look like the following:

```

ratings = Rating.objects.filter(user_id=user_id)
sim_users = Rating.objects.filter(movie_id__in=ratings.values('movie_id')) \
    .values('user_id') \
    .annotate(intersect=Count('user_id')).filter(intersect__gt=min)

```

This is a slow query, but considering that we just cut away a big chunk of users, which wouldn't have been similar to the current user, maybe it was worth the wait. And with this list, we can now say that we only want to look at the 100 most similar users. Or the users that have at least a certain number of similar items. Looking around for advice I found that Michael D. Ekstrand and friends from GroupLens teams suggest using somewhere between 20-50 users. My feeling is that is a high number on real data, but on the data set we have, it might be a good number. I dare you to test it. We will look into how to implement the Pearson similarity, which we discussed above. The following method will compare to users and calculate their similarity

Pearson Similarity functions Listing 7.8: Pearson method in \recommender\views.py

```

def pearson(users, this_usr, that_usr):
    if this_usr in users and that_usr in users:
        this_usr_avg = sum(users[this_usr].values()) / len(users[this_usr].values()) ①
        that_usr_avg = sum(users[that_usr].values()) / len(users[that_usr].values()) ①

    all_movies = set(users[this_user].keys()) & set(users[that_user].keys()) ②

    dividend = 0
    divisor_a = 0
    divisor_b = 0
    for movie in all_movies:
        if (movie in users[this_user].keys()
            and movie in users[that_user].keys()):
            nr_a = users[this_user][movie] - this_user_avg ③
            nr_b = users[that_user][movie] - that_user_avg ④
            dividend += (nr_a) * (nr_b)
            divisor_a += pow(nr_a, 2)
            divisor_b += pow(nr_b, 2)

    divisor = Decimal(sqrt(divisor_a) * sqrt(divisor_b))
    if divisor != 0:

```

```

    return dividend/Decimal(sqrt(divisor_a) * sqrt(divisor_b))      5
    return 0               6

```

- 1 Find the users average rating.
- 2 merge the two movie sets into one.
- 3 Pearson only looks at items that has been rated by both users.
- 4 Normalize user ratings by subtracting the mean.
- 5 Finally put everything together and calculate Pearson
- 6 return zero, if the divisor was always zero.

7.4.1 Implement the similarity in MovieGEEKs site

To make it easier to play around with the similarities, we will have a look at how it is implemented in the MovieGEEK site admin part.

Did you have a look at the MovieGEEK site, it has an analytics part, that in turn contains a page for each user_id, it can be found using the following url: (assuming that you have it on localhost:8000)

<http://localhost:8000/analytics/user/100>

If you want to look for user with id 100. User 100 likes many different genres, for example adventure, animation and thrillers, as the chart shows. Have a look for yourself in figure 7.12 or on the site.

User Profile id: 100 (in cluster: [14](#))

Average rating: 7.33 / 10

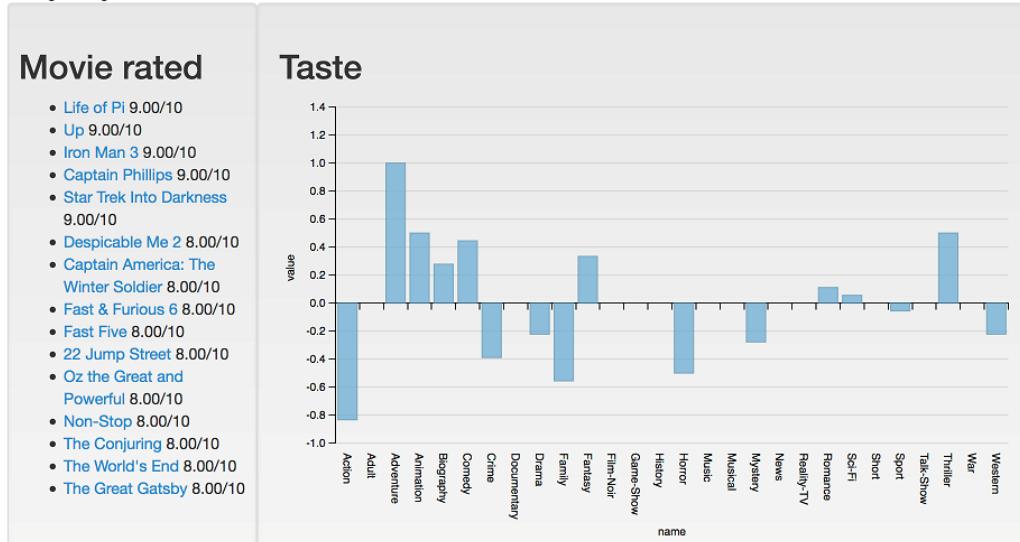


Figure 7.1217 Top of the user profile page of user 100

Our task in this section is that we want to implement the next section on the page. The one that shows similar users. Which you can see in Figure 7.13.

Users similar to user 100

Jaccard sim

- [100:1](#)
- [26790:0.27](#)
- [51171:0.26](#)
- [49813:0.26](#)
- [32605:0.25](#)
- [21906:0.24](#)
- [25492:0.24](#)
- [24091:0.24](#)
- [49621:0.24](#)
- [43781:0.23](#)

Pearson sim

- [41244:1](#)
- [100:1](#)
- [32303:0.99](#)
- [31805:0.99](#)
- [18449:0.98](#)
- [40537:0.97](#)
- [7291:0.97](#)
- [40636:0.96](#)
- [15738:0.96](#)
- [34127:0.95](#)

Figure 7.1318 Similar users to user 100, calculated to the left by Jaccard and Pearson on the right.

In the MovieGEEKs site, the similarity is considered part of the recommender system, so I have added a `similar_users` method to the recommender API. The `similar_users` requires a

user_id and a type. The type is added to enable you to extend it with other types of similarity calculations easily. I shall leave it up to the interested reader to see the configuration around it and skip directly to the Python code of the method.

Listing 7.9: similar_users in recommender/views.py

```
def similar_users(request, user_id, type):

    min = request.GET.get('min', 1)                                1
    ratings = Rating.objects.filter(user_id=user_id)               2
    sim_users = Rating.objects.filter(movie_id__in=ratings.values('movie_id'))\ 
        .values('user_id') \
        .annotate(intersect=Count('user_id')) \
        .filter(intersect__gt=min)                                 3

    dataset = Rating.objects.filter(user_id__in=sim_users.values('user_id')) 4
    users = {u['user_id']: {} for u in sim_users}                  5

    for row in dataset:                                           6
        if row.user_id in users.keys():
            users[row.user_id][row.movie_id] = row.rating

    similarity = dict()
    switcher = {                                                 7
        'jaccard': jaccard,
        'pearson': pearson,
    }

    for user in sim_users:
        func = switcher.get(type, lambda: "nothing")           8
        s = func(users, int(user_id), int(user['user_id']))     9
        if s > 0.2:                                            10
            similarity[user['user_id']] = s                      11

    data = {                                                       12
        'user_id': user_id,
        'num_movies_rated': len(ratings),
        'type': type,
        'similarity': similarity,
    }
    return JsonResponse(data, safe=False)
```

- ① you can also add a min overlap required.
- ② Get all the current users ratings
- ③ Based on the current users ratings, retrieve all users who has also rated one or more of those films.
- ④ Get the ratings of all the users who has overlapping ratings with the user.
- ⑤ Extract all the user_ids
- ⑥ Build a user rating matrix
- ⑦ A sneaky way of doing a case statement in python, if you want to add another similarity method, you simple add the name and the name of the method.
- ⑧ Now iterate through all the users, and
- ⑨ Get a reference to the func which were described in the input, and execute it
- ⑩ Check if the similarity is more than 0.2. Change this to require more or less similarity.
- ⑪ Add user to the list of similar users.
- ⑫ return as JSON.

The Pearson method is described in detail above, while the Jaccard is shown below, please refer to the theory earlier in the chapter for a detailed description.

Listing 7.10: Jaccard method in recommender/views.py

```
def jaccard(users, this_user, that_user):
    if this_user in users and that_user in users:
        intersect = set(users[this_user].keys()) \
                    & set(users[that_user].keys())
    union = set(users[this_user].keys()) | \
            set(users[that_user].keys())
    return len(intersect)/Decimal(len(union))
else:
    return 0
```

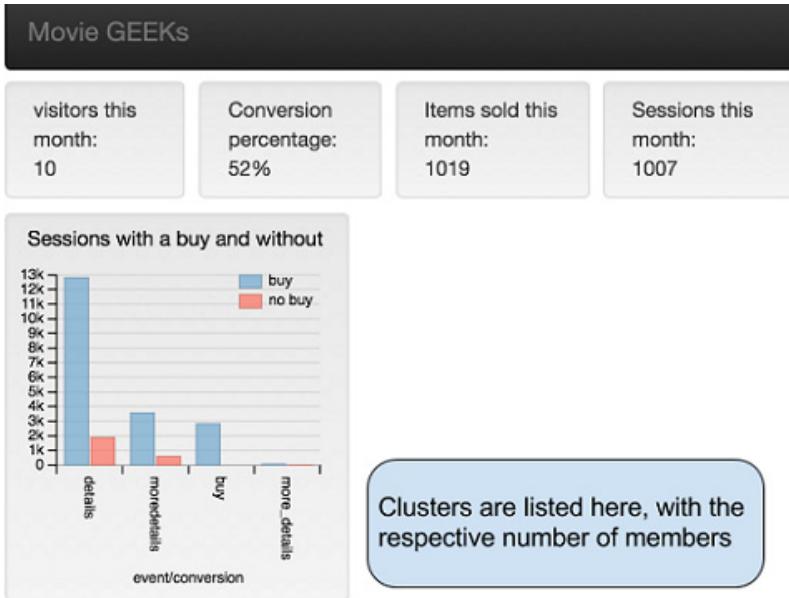
- ➊ calculate the intersection between the two users.
- ➋ calculate the union between the two users.
- ➌ return the Jaccard correlation.

I did a small test to see if the Pearson similarity and k-means clustering worked the same way, by sampling a few of the users, which I found in the list of similar users to user 2, with respectively 0.87 and 0.85 similarity, and they were all in cluster seven. You can't prove that it works for all just by looking at one example, but it does give a good indication that the implementation is consistently either wrong or correct.

The user similarities that we looked at here, doesn't require any training. In Chapter 8 we will look at item-similarity, and there we will calculate all the similarities beforehand.

7.4.2 Implement the clustering in MovieGEEKs site

Using clustering algorithm in our site will be a bit of a leap of faith for now. We will implement therefore implement the solution using the clustering algorithm that is part of the Scikit-learn library, it's also a k-means algorithm, but supposedly its faster, and better tested than our little example, so we will go with that. We will add the Clusters to our Analytics part of the MovieGEEKs site in two places, firstly on the main page (<http://localhost:8000/analytics/>). Of course, no clusters will be shown before we have calculated them.



Top 10 content

- Teenage Mutant Ninja Turtles (42)
- Hail, Caesar! (38)
- The Huntsman: Winter's War (38)
- Now You See Me 2 (33)
- Kung Fu Panda 3 (32)

Clusters

- 0(1)
- 1(321)
- 2(3)
- 3(10)
- 4(491)
- 5(300)
- 6(510)

Figure 7.14 Showing the loading page of the analytics dashboard with the list of clusters.

The script implemented for the MovieGEEKs is as follows, it does the same as the script we saw above, only now it will load data from the database, calculate the kmeans and then save a row in the Cluster table for each user_id, with its corresponding cluster id.

Listing 7.11: UserClusterCalculator script in \builder\user_cluster_calculator.py

```
class UserClusterCalculator(object):

    def load_data(self):
        print('loading data')
        user_ids = list(
            Rating.objects.values('user_id')
                .annotate(movie_count=Count('movie_id'))
                .order_by('-movie_count'))
        content_ids = list(Rating.objects.values('movie_id').distinct())

```

1

2

```

content_map = {content_ids[i]['movie_id']: i
              for i in range(len(content_ids))}                                     ③
num_users = len(user_ids)
user_ratings = dok_matrix((num_users,
                           len(content_ids)),
                           dtype=np.float32)                                         ④

for i in range(num_users):
    # each user corresponds to a row, in the order of all_user_names
    ratings = Rating.objects.filter(user_id=user_ids[i]['user_id'])
    for user_rating in ratings:
        user_ratings[i, content_map[user_rating.movie_id]] = user_rating.rating
print('data loaded')                                                 ⑤

return user_ids, user_ratings
}

def calculate(self, k = 23):
    print("training k-means clustering")

    user_ids, user_ratings = self.load_data()

    kmeans = KMeans(n_clusters=k)
    clusters = kmeans.fit(user_ratings.tocsr())                                ⑥
                                                                           ⑦

    plot(user_ratings.todense(), kmeans, k)

    self.save_clusters(clusters, user_ids)

    return clusters

def save_clusters(self, clusters, user_ids):
    print("saving clusters")
    Cluster.objects.all().delete()
    for i, cluster_label in enumerate(clusters.labels_):                      ⑧
        Cluster(
            cluster_id=cluster_label,
            user_id=user_ids[i]['user_id']).save()                                 ⑨

if __name__ == '__main__':
    print("Calculating user clusters...")

    cluster = UserClusterCalculator()
    cluster.calculate(23)

```

- ① Get out all the user_ids from the ratings table.
- ② Get all the content_ids also from the ratings table.
- ③ Create a mapping between the content_ids and a list of integers, to make it work with the sparse matrix implementation we use.
- ④ Create an instance of dok_matrix (Dictionary of keys matrix)⁴⁰, with the dimensions according to number of users and content.

⁴⁰ http://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.dok_matrix.html

- 5 Iterate through all the users and add data to the matrix.
- 6 Create an instance of KMeans clustering algorithm.
- 7 Do the magic (clustering)
- 8 Delete all clusters already saved in the database, to make room for the new ones.
- 9 save the clusters.

This script can be run from the command line (or directly from PyCharm, which I am using). From the command line simply write:

Listing 7.12: Running the clustering

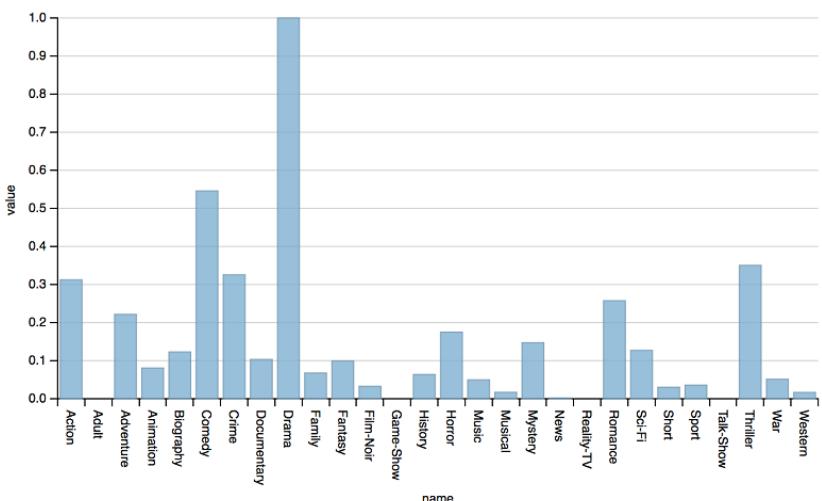
```
python -m builder.user_cluster_calculator
```

On a MacBook from 2014, it takes a couple of hours, so go and get some exercise and food, before moving on.

Besides the view on the loading page of the analytics, you can also click on one of the clusters, for example the first one and you will get the view shown in figure 7.15. It shows a chart of the normalized taste of all the cluster members, plus a list of all its members.

Cluster 3

Cluster has 35 members



```
2365 6087 7527 10041 10049 10311 13950 14587 15291 16321 17773 20732 21104 21520 22445 25239 26526 30882
31000 32491 32541 32922 33736 35319 36466 40187 40864 40992 41561 41629 43510 44751 47288 50612 51564
```

Figure 7.15 Screen shot from the analytics part showing a cluster.

It is a chapter in itself to describe and discuss how you understand and verify if your clustering algorithm works to your advantage, in the following chapter we will use it to narrow down the search for similar users, so we can calculate recommendations faster.

7.5 Summary

Similarity and distance are everything in recommender systems and indeed also in many machine learning algorithms. A lot of work for a data scientist will often go into transforming data from categories or text into a format that can be used to calculate similarities. It is therefore important to have a good mental picture of similarity functions.

In this chapter, we have talked about data which can be visualized, like 2-dimensional quantitative data. It is great for showing examples but quite rarely what you see in real-life. However my advice is get well acquainted with the 2d data similarity examples we have done here, as it will help you in the future to better understand more complex situations.

- Measuring similarities between sets, can be used to calculate similarity between users where the data is transactional like buy data, or like data.
- Quantitative data is interesting when we talk about users who have given us ratings, either implicitly or explicitly.
- The L_p norms can be used on quantitative data, and are also the starting points for talking about other types of correlation methods like the Pearson correlation and Cosine similarity, which look very much alike but has different interpretations.
- K-means clustering are great in toy example, but be always be careful about checking the results. As we saw it can easily end in a sub optimal state, which would create strange recommendations to users. The example we looked at came with a warning reminding readers that examples are constructed to explain the algorithm, but hides the fact that many algorithms are quite hard to make work according to the executors wishes, so consider yourself warned.

8

Collaborative Filtering in the Neighborhood

Collaborating makes things easier, so let's collaborate our way through this chapter:

- You'll start out revisiting the rating matrix.
- You will look at the theory behind Collaborative filtering.
- Collaborative filtering is done in several steps, you will look at each and learning about the choices that needs be taken.
- As always you will learn about how Collaborative filtering is implemented in MovieGEEKs.

In this chapter, we will introduce collaborative filtering and then go into detail about the branch of it called neighborhood-based filtering. Collaborative filtering is an umbrella of methods. What unites them is their selection of data. They only use ratings (implicit or explicit) as the source for creating recommendations. We will dedicate two chapters for collaborative filtering, this one and chapter 10. In chapter 10 we will look into learned models using matrix factorization to find hidden features, also better known as latent features. In between the two, in chapter 9, we will look at content based filtering⁴¹.

In this chapter, you will learn about the history of recommenders and see different ways collaborative filtering has been used. The core of collaborative filtering we are looking at in

⁴¹ It's not completely true, because we also use collaborative filtering in chapter 12 about hybrid recommenders as one of the feature recommenders, and again in chapter 13 on ranking algorithms. But they are not the focus of the chapters.

this chapter is called neighborhood-based filtering and is based on similarity between users and between items, calculated with functions like the ones we covered in Chapter 7.

So far, we have only created some simple, non-personalized recommenders. It is now time to get close and personal with personalized recommendations. The recommendations we have created so far was based on auto generated collected behavior; from now on we will only use the real rating dataset called MovieTweetings⁴² and base the recommendation building on it. I recommend checking the dataset out at [github](https://github.com/sidooms/MovieTweetings), and familiarize yourself with it.

The collaborative filtering algorithm is quite simple. There are only a few things that need to be in the pipeline to make it produce recommendations. In each of these pipeline steps you have a list of choices that will affect the outcome. We shall look at each of these steps in detail and try to make sense of it all.

When we are finished with this chapter we will know how to implement the Amazon recommender algorithm the item-item collaborative filtering algorithm. At least the one that they published back in 2003⁴³. I would be surprised if they haven't come up with something different now. The algorithm was used to produce the "Recommended for you" page. Mine is shown in Figure 8.1; as you can see I have been buying books on statistics and Django. The overall idea is to find items that are rated similarly to the items you have already rated or bought.

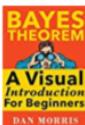
⁴² <https://github.com/sidooms/MovieTweetings>

⁴³ G. Linden et al. *Amazon.com recommendations: item-to-item collaborative filtering* <http://ieeexplore.ieee.org/abstract/document/1167344/>

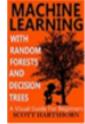
Your Amazon.co.uk > Recommended for you
 (If you're not Kim Falk, click here.)

These recommendations are based on items you own and more.

view: All | New Releases | Coming Soon More results 

1.  **Bayes' Theorem Examples: A Visual Introduction For Beginners**
 by Dan Morris (2 Oct. 2016)
 Average Customer Review: ★★★★☆ (11)
 Available for download now
Kindle Price: £1.99

I own it Not interested ★★★★☆ Rate this item
 Recommended because you purchased **Statistical Snacks** and more ([Fix this](#))

2.  **Machine Learning With Random Forests And Decision Trees: A Visual Guide For Beginners**
 by Scott Hartshorn (12 Aug. 2016)
 Average Customer Review: ★★★★★ (8)
 Available for download now
Kindle Price: £1.99

I own it Not interested ★★★★☆ Rate this item
 Recommended because you purchased **Statistical Snacks** and more ([Fix this](#))

3.  **Fluent Python**
 by Luciano Ramalho (20 Aug. 2015)
 Average Customer Review: ★★★★★ (19)
 In stock
RRP: £39.99
Price: £28.29
36 used & new from £14.52 [Add to Basket](#) [Add to Wish List](#)
 I own it Not interested ★★★★☆ Rate this item
 Recommended because you added **Two Scoops of Django 1.11** to your Shopping Basket and more ([Fix this](#))

Figure 8.119 My "Recommended for you" page at Amazon.

Neighborhood collaborative filtering based algorithms were the first algorithms to be categorized as a recommender systems algorithm, so we need to start with some history.

8.1 What is collaborative filtering

A history lesson

Most people consider themselves unique and, therefore, don't like to be segmented into a particular type. But, that is exactly what using collaborative filtering to calculate recommendations is all about. In all its simplicity, collaborative filtering is recommending a list of items for you. The list being created based on people who like the same thing as you, but who also like something that you haven't consumed yet.

8.1.1 When information became collaboratively filtered

Our story starts around 1992 at the Xerox PARC (Palo Alto Research Center), where they realized that the number of emails being sent around had exploded and "*which is resulting in users being inundated by huge stream of incoming documents*"⁴⁴. I can't help thinking that in 1992 they truly had seen nothing yet, but as in so many other cases the Xerox PARC was ahead of its time, maybe also in information overload.

The mail system they built was based on the assumption that you would always have a few users that would read everything immediately and would endorse interesting things, while most users would just read what they thought look intriguing. The system was called Tapestry; this is a name you'll often read in recommender system literature.

Two years later, the GroupLens project created "an open architecture for collaborative filtering of netnews."⁴⁵ GroupLens wanted to solve the same problem of information overload and wanted to enable people to annotate newsgroup messages with a rating. This time, the system built on the assumption that people who agreed with the ratings in the past were likely to agree on it again. Between the two of them, Xerox and the GroupLens Group laid down the foundation for most of what we know as recommenders today. The following will largely be a description of what GroupLens did back then, along with the improvements that have been suggested since then.

8.1.2 Helping each other

So, the assumption is that together we can be better, together we will better understand each other. Sounds beautiful and a bit cheesy like ending of an epic Hollywood film, but this is the idea.

We need to assume that people principally keep their taste over time and that if you agreed with somebody in the past, then you agree with them in the future also.

Let's try to be a bit more concrete before we dig into the theory of it all and how to actually calculate it. In chapter 6, we looked at recommendations that were based on what people had bought in the past by looking at their shopping baskets; now we will concentrate on the user – *if the user was a shopping cart what would be in it?*

Bookshops and libraries often have posters saying, "If you liked this popular book X, then you should also try (maybe not so popular) book Y." These posters are directed toward a large group of people and often work well; they are similar to a filtered chart. What we want to do is create individualized lists or at least lists for tiny groups of like-minded users. We don't want to print them out and hang them on walls, but rather create and present them instantly when a user arrives.

⁴⁴ D. Goldberg et al. *Using collaborative filtering to weave an information tapestry* (1992).

<http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.104.3739>

⁴⁵ <http://ccs.mit.edu/papers/CCSWP165.html>

Neighborhood-based filtering can go two ways. We could find users with similar film tastes as yours and then recommend films they have liked but that you haven't seen; this is user-based filtering (start in the upper left corner then go right and down in figure 8.2). Alternatively, we can find items similar to items that you have already liked (start in the upper left corner, then go down and to the right in figure 8.2). Both similarities between users and between items are calculated based on the ratings.

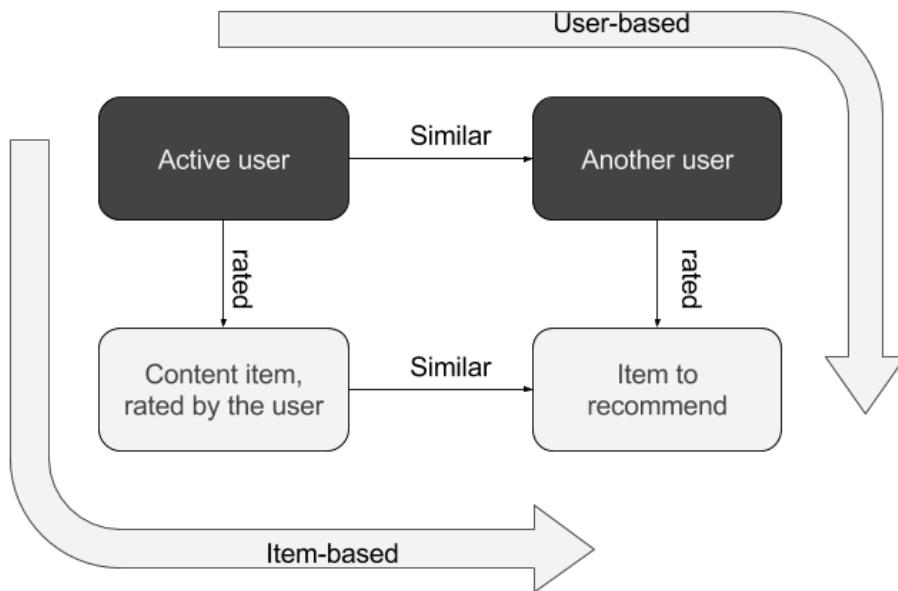


Figure 8.220 The two ways of neighborhood-based filtering. Either via similar users or via similar items to items the active user liked.

The user-based path means looking at similar users. Take my good friend Thomas for example. I have known him for many years and I am pretty sure he likes similar movies. So, if I want to go to the cinema, I can text him and ask for advice. If I had a group of friends with similar tastes to mine, I could ask all of them and then use all of their responses. The group would collaboratively filter the current selection of movies for me, in telling me which are recommendable.

For example, say you have watched the new Star Trek movie, and you want to watch something similar. You ask all of your friends what they thought about *Star Trek* and if they can give you other films they liked the same. The story could go that I asked Helle who I know likes science fiction and she can tell me that she also liked *Rouge One*, while Pietro has no love for science fiction, so he naturally hated it, and promised me that he would never waste

any time on another film like *Rouge One*. Having these two testimonies means that I can deduce that the two films are similar in that one person likes both and one doesn't like any of them. Since I liked *Star Trek* then I probably also like *Rouge One*. In this way, my friends have collaboratively helped me find the next film I can go and see in the cinema (was I not spending all my evenings writing this book). So, in short either you find similar users to the active user, and then recommend the films they liked, and the active user hasn't watched, or you find the liked items of the active user and find similar items to those, and recommend them. To make all this work we are back at the rating matrix as a means to describe user's preferences.

8.1.3 The rating matrix

A way to represent users' tastes is to list all the content they have in some way expressed an opinion about. Usually, this data is kept in a user-item matrix, which we talked about back in chapter 7, and which we will be quickly revisiting here.

The word "matrix" is just a fancy word for a table with numbers, like the one shown in Table 8.1. Each cell indicates a user's sentiment toward a content item. Usually, a high rating like 5 or 10 is a good rating (depending on what rating scale you are using), and a rating close to zero is a bad rating or a dislike, but that is just a question of definition.

What we want to calculate are predictions for each of the empty cells in this matrix – numbers that correspond to a particular user's predicted future sentiment toward specific content items – using the data already present in the matrix. Does that mean that we want all the data to be present? Not really, because that would indicate that the user has rated or consumed all content you can offer already, which of course is good, but then they have to go somewhere else to get more. On the other hand, if it is too empty, then we will have the cold-start problems as we talked about at length in chapter 6.

Table 8.1: Example of a rating matrix. Notice that Sara has not rated *Ace Ventura*, and Jesper has not rated *Braveheart*.

	Comedy	Action	Comedy	Action	Drama	Drama
Sara	5	3		2	2	2
Jesper	4	3	4		3	3
Therese	5	2	5	2	1	1
Helle	3	5	3		1	1

Pietro	3	3	3	2	4	5
Ekaterina	2	3	2	3	5	5

So how does this fit into what I said about finding similar users and items? If we look at the table, we can see Sara and Therese have more or less similar taste, so if we should find a good movie to recommend Sara we could choose a movie Therese likes and Sara hasn't seen. Or we could say Sara liked *Men in Black*, and find which films have been rated similar to it, if you look at *Ace Ventura* the same people liked it (Jesper and Therese) and the same people (Helle, Pietro and Ekaterina) didn't, so that can be considered similar. This is simplified, and you should remember that normally there are more empty cells than filled ones so it requires a bit more to do it. So, what does it take exactly to make a program do this.

8.1.4 The collaborative filtering pipeline

When talking about Machine Learning and Predictive applications you typically talk about a pipeline, which is a serialized pipe (possibly parallelizable) of things that need to be calculated in a certain order before predictions can be made.

To give an overview of what steps needs to be taken, we shall look at the pipeline shown in figure 8.3. You learned in Chapter 7 that there are several ways to calculate similarity. One of these can be used to calculate the similarity, which is the first step and shown at **1** in the figure, later we will discuss which function should be used. Using one of these will enable us to create a list of the active user's similarity with all the other users. **2**, orders the other users. In **3**, select a neighborhood to be used to calculate the predictions, again there are different ways you can decide which is the neighborhood, but for now just think of it as a close group of users, which are similar to the active user. Later in this section, we will look at the ways this can be done. At **4**, the similarities for the users in the neighborhood is used along with these users ratings of the item to predict. Here the predicted rating is 3.48. Predicted ratings can either be used as is, or we can calculate lots of ratings, then order these and finially return the top-N predicted ratings and we have a top-N recommendation.

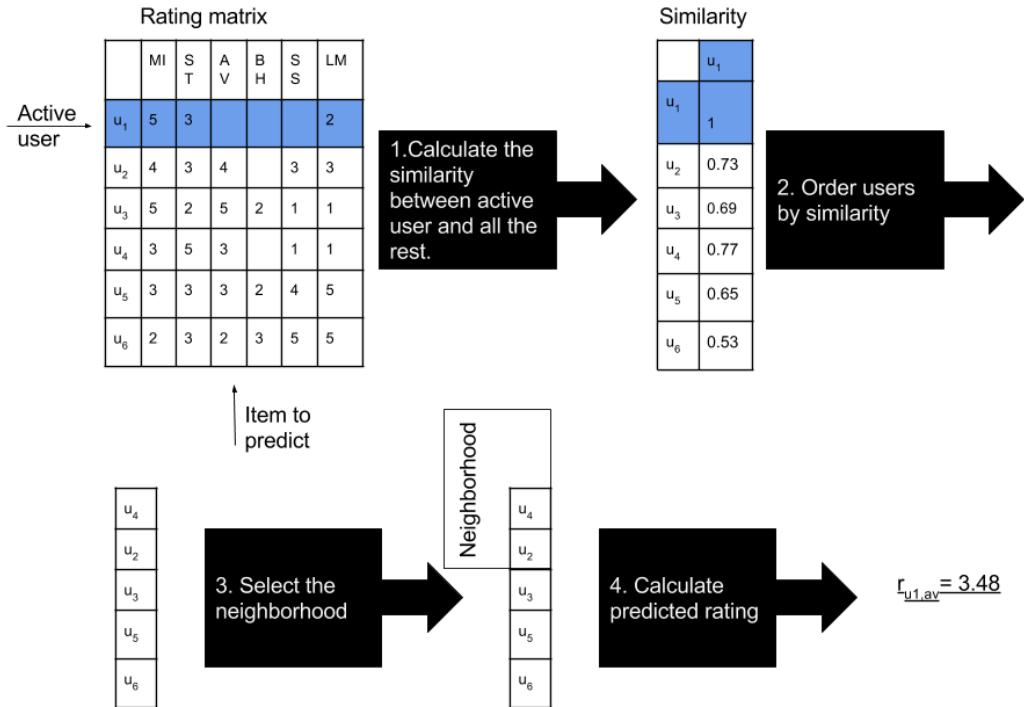


Figure 8.3 Pipeline for user-based neighborhood-based filtering

The goal of most systems is to do as much as possible before the user arrives and demands recommendations. We will also look at what can be done.

8.1.5 User-user collaborative filtering

If we start looking at the first collaborative filtering algorithm, which finds similar users and uses that to calculate recommendations.

An average user doesn't have a lot of item ratings, which means that adding a new one can change the system's calculation of the user's taste. Therefore, it is considered unwise to do pre-computation of which users are similar. Items, on the other hand, are considered more stable in the sense that same type of people like the same type of items, and studies have shown that you can pre-calculate similarities. And that is king when you are talking about a catalog the size of the Amazon (You can think about both the size of the Amazon River or Amazon the internet store!). If you look at figure 8.2 again, you should consider that we want to calculate as few similarities as possible, so if you have many more users than items, then you should go for item based similarities, otherwise, user-based filtering would be the more economical.

The reason why anybody is still talking about user-user based filtering is that it is considered a better way to give recommendations. Because if you do item to item you will find items similar to what you have already rated, but it won't provide the serendipity that similar users could provide, where the hope is that your data will connect you with users with different peculiarities in their taste and therefore at times provide surprising but good recommendations.

On the other hand, if we want to explain why we give these recommendations, then Item based filtering makes this task very easy, as the system can simply say you get a recommendation for movie Y because you liked movie X which is similar to movie Y. It is easy to justify the recommendations, while if you use user-based filtering it requires a bit more ingenuity to explain why the recommendations are shown while keeping the privacy of the other users.

8.1.6 Data Requirements

Does the data match the needs of the collaborative filtering?

I have noted on several occasions that there are no clear rules on which recommender algorithm you should use. To a far extent, that is still true. Nonetheless, I still have some advice to give you before you venture into the exciting world of collaborative filtering.

To calculate recommendations, the data needs to be well connected.

1. If no users have rated content, then no recommendations will be made.
2. Users who don't have overlapping taste with other users won't receive good recommendations. A way to calculate this, is by first finding all the content items which has been rated by several users (more than 2 user at least). Then calculate the number of users who are connected to one or more of these content items. That is the users that will receive recommendations, the rest won't.
3. A rule of thumb is that the system needs to have around ten times as many content items as users. But take that with a grain of salt, and remember it depends on what type of content you are dealing with. Music sites need more than ten times as many songs, while travel agencies might need fewer content items.

The good thing about collaborative filtering is that your system doesn't need any domain knowledge at all. But remember that you do to create a good recommender.

8.2 Calculate Recommendations

So far, we looked at both user and item-based filtering. We will continue talking about user-based filtering, but the implementation that we will end up with is using item similarity since the data set we have circa 45.000 users and only 25.000 items. The pipeline is a tiny bit different, but you still have the same steps (as we saw in figure 8.3) only you are just looking at items instead of users. In the following, we will have a look at each of the steps shown in figure 8.4.

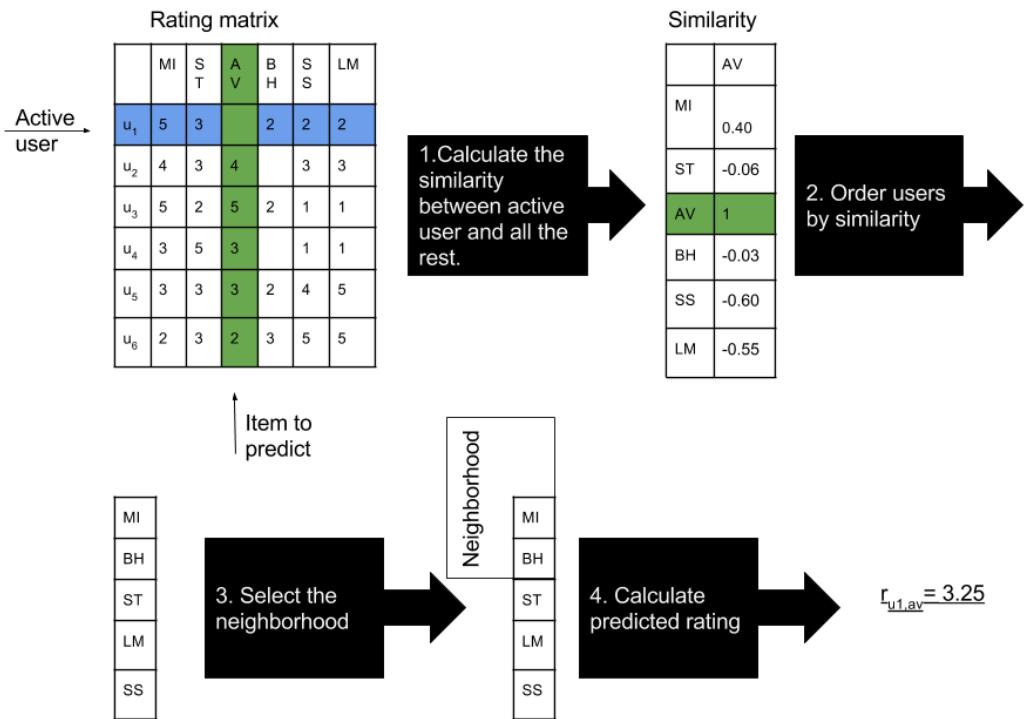


Figure 8.4 showing the item-based filtering pipeline.

8.3 Calculating the similarities

The first thing we need to do is to settle on a similarity function, which we learned about in chapter 7. There are different schools of thought about which similarity function to use. The implementation we will look at, uses Cosine Similarity learned in the previous chapter. It will provide a matrix of similarities which for each item will provide a list of similar items.

8.4 Amazon's algorithm to pre-calculate item similarity

Item similarity is said to be quite stable, and you can therefore calculate the similarities beforehand or offline as they say. Amazon was one of the first, and probably biggest users of this algorithm and they were kind enough to publish a paper describing their method. Greg

Lindens, the builder of much of Amazon.com's early recommendation system, described a pseudo-algorithm for how to do item-item collaborative filtering⁴⁶.

```

For each item in product catalog, I1
  For each customer C who purchased I1
    For each item I2 purchased by customer C
      Record that a customer purchased I1 and I2
  For each item I2
    Compute the similarity between I1 and I2
  
```

Using this algorithm, you will end up with a dataset, where you can look up similar items for the current item, and that makes it quite fast when you calculate the predictions. The procedure is explained in more detail in Amazons patent from 1998⁴⁷. Which illustrate the algorithms as shown in Figure 8.5

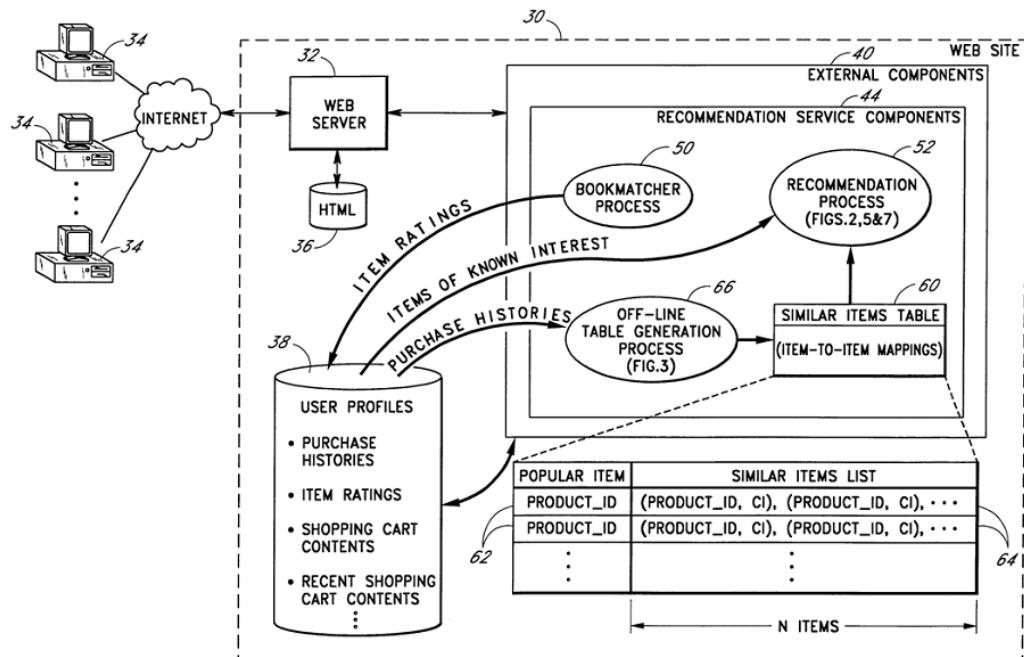


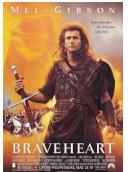
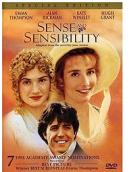
Figure 8.5 Amazon patented its item-item collaborative recommendation algorithm, this is one of the diagrams explaining the algorithm

⁴⁶ Linden et al."Amazon.com Recommendations Item to Item Collaborative Filtering" <https://www.cs.umd.edu/~samir/498/Amazon-Recommendations.pdf>

⁴⁷ <https://www.google.com/patents/US6266649>

Let's do a little example.

Table 8.2: (same as table 8.1) Example of a rating matrix. Notice that Sara has not rated Ace Ventura, and Jesper and Helle has not rated Braveheart.

						
	Comedy	Action	Comedy	Action	Drama	Drama
Sara	5	3		2	2	2
Jesper	4	3	4		3	3
Therese	5	2	5	2	1	1
Helle	3	5	3		1	1
Pietro	3	3	3	2	4	5
Ekaterina	2	3	2	3	5	5

Take the table 8.2 above, according to the algorithm; we should go through all the movies and look at each customer. All customers have rated MIB, so we will have to look at each user in turn. The first user is Sara, who also rated *Star Trek* (ST), *Braveheart* (B), *Sense and Sensibility* (SS) and *Les Miserables* (LM). So we should add that to the list items rated together with MIB, so we have:

MIB: [ST, B, SS, LM]

Next, we have Jesper, who rated things already in the list plus *Ace Ventura* (AV). An item should only be added once, so we will now have the following list:

MIB: [ST, B, SS, LM, AV]

This is going to be a long story if we will do it for all, so we will skip ahead, but you as a dutiful reader should continue and do the model for all of them. For the rest this would be the result:

MIB: [ST, B, SS, LM, AV]

ST: [MIB, B, SS, LM, AV]

B: [MIB, ST, SS, LM, AV]

SS: [MIB, ST, B, LM, AV]

LM: [MIB, ST, SS, B, AV]

AV: [MIB, ST, B, SS, LM]

With these lists we can calculate similarities for each of the elements in the list for MIB. Again, we will just do it for the first one, and then leave it as an exercise for you to calculate the rest.

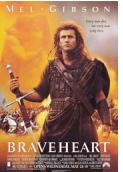
We use the adjusted cosine similarity (the one that normalizes the item ratings based on the users average instead of the items) function to calculate the similarity. To make the calculations fit we will again define: $nr_{i,u} = r_{i,u} - \bar{r}_u$

$$\text{Sim}(\text{"MIB"}, \text{"ST"}) = \frac{\sum_u nr_{MIB,u} nr_{ST,u}}{\sqrt{\sum_u nr_{MIB,u}^2} \sqrt{\sum_u nr_{ST,u}^2}}$$

I have to admit that I am a bit of trouble here, I tried to write out the summations, but either the example got so small that it was too obvious, or I couldn't fit the calculations on the page. But let's do small steps and then maybe it will work.

First, we want to normalize the table above. This is again done by calculating the average rating for a user and subtract it for its ratings (this is just one way of doing it of course, in the code we will use something a bit more complex.):

Table 8.3: The same table as table 8.2, only now we have normalized the ratings. According to the user's average

						
	Comedy	Action	Comedy	Action	Drama	Drama
Sara	1,80	1,80		-1,20	-1,20	-1,20
Jesper	0,20	1,20	0,20		-0,80	-0,80
Therese	2,17	0,17	2,17	-0,83	-1,83	-1,83
Helle	0,40		0,40	2,40	-1,60	-1,60
Pietro	-0,33	-0,33	-0,33	-1,33	0,67	1,67
Ekaterina	-1,33	-0,33	-1,33	-0,33	1,67	1,67

With the normalized ratings, we can calculate the similarity between MIB and ST. To do that we will look at each user in turn and sum it. So, we will multiply each rating pair for each user, and add them together (I have pointed out Jesper's contribution to the equation)

Jesper's ratings

$$\begin{aligned}
 & \frac{(2.2 * 0.2) + (0.6 * -0.4) + (2.33 * -0.67) + (0.4 * 2.4) + (-0.33 * -0.3) + (-1.33 * -0.3)}{\sqrt{2.2^2 + 0.6^2 + 2.33^2 + 0.4^2 + -0.33^2 + -1.33^2} * \sqrt{0.2^2 + -0.4^2 + -0.67^2 + 2.4^2 + -0.33^2 + -0.33^2}} \\
 & = \frac{0.1467}{\sqrt{3.559} * \sqrt{2.574}} = \boxed{0.016} \\
 & \quad \text{Jesper's rating of MIB} \qquad \qquad \qquad \text{Jesper's rating of ST}
 \end{aligned}$$

Similarity

"Hmm," you might be thinking. What does this tell me? Well, we know that the adjusted cosine similarity function returns results between -1 and 1. And basically, you could interpret it like how many of the people who has rated both films have given them both above average or under. And if you look at ST and MIB the numbers in the table 8.3 above then you can see that Sara, and Helle gave both films above averages, while Pietro and Ekaterina below. And then Jesper and Therese like one more than average and the other one no. And it is exactly Jesper and Therese who are the trouble makers here, but also Sara and Helle give it very different ratings, so none of the users really agree on the two movies, and that is also visible in a similarity of 0.016. I have calculated all of the similarities in the following:

Table 8.4 Similarity matrix between the 6 movies. The negative similarities are highlighted with red, while the positive ones are colored green.



	Comedy	Action	Comedy	Action	Drama	Drama
Sara	1	0.63	1	-0.21	-0.88	-0.83
Jesper	0.63	1	0.35	-0.47	-0.64	-0.62
Therese	1	0.35	1	0.01	-0.89	-0.83
Helle	-0.21	-0.47	0.01	1	-0.23	-0.32
Pietro	-0.88	-0.64	-0.89	-0.23	1	0.96
Ekaterina	-0.83	-0.62	-0.83	-0.32	0.96	1

To get a feel for how the similarity is distributed, it is a good idea to have a look one in each end of the scale and one in the middle:

- **Close to 1:** LM and SS, The LM and SS are interesting because you can see that they are dissimilar to all except themselves. They have high similarity since all users agree on rating both either above or under. But then why is it not 1? that is because they are not exactly the same. If they were completely the same, it would be one, as is the case with AV and MIB.
- **Close to -1:** SS and AV are the most dissimilar between the movies. Which also makes sense, I could imagine that most people who like *Ace Ventura* don't like *Sense and Sensibility* and the other way around.
- **One close to 0:** We just calculated ST and MIB

When I calculated this in python, I could have added an if-statement which would only indicate similarity if the similarity function return values above zero. That made the lists above much shorter, and easier to overview. But in some cases, it can also be worth to use the negative similarities or the dissimilarities. For example could you consider recommending *Ace Ventura* to people who didn't like *Sense and Sensibility*

BEWARE OF THE “1 OR 2 ITEMS IN COMMON” PROBLEM

In the algorithm above, if an item only has one rating, the average rating is equal to the one rating. That means that $nr_{a,i}$ will be zero, which will make the similarity function undefined. Next, if you have one user who is the only one who rated two items, then the similarity is one, which is the highest similarity you can get. So you should be careful leaving the algorithm to calculate similarity between users that has too few items in common.

Imagine that it is only Helen who has watched both MIB and ST, and she has rated them quite differently, but the function will say they are a top match:

$$\frac{2.17 * 1.7}{\sqrt{2.17^2} \sqrt{1.7^2}} = 1$$

Therefore, always require a minimum of overlapping users, overlapping in the sense the number of users who has rated both films. This means that you should not use collaborative filtering with users who has only rated one or two items only.

On the other hand, if there are too many overlapping users it can also be difficult to find similarities, the recommendations that will be produced are too general, if everybody likes some items then it will just turn into a chart.

So, remember it is a tradeoff. Narrow the number of users, and you risk not finding any content to recommend. Make it too wide, and you risk that the recs contain content that is outside of the user's taste. From a practical point of view have a look at figure 8.6, which shows how many users have only rated one item and so on. In fact, almost 20.000 out of the 45.000 users has only rated one movie in the MovieTweetings dataset.

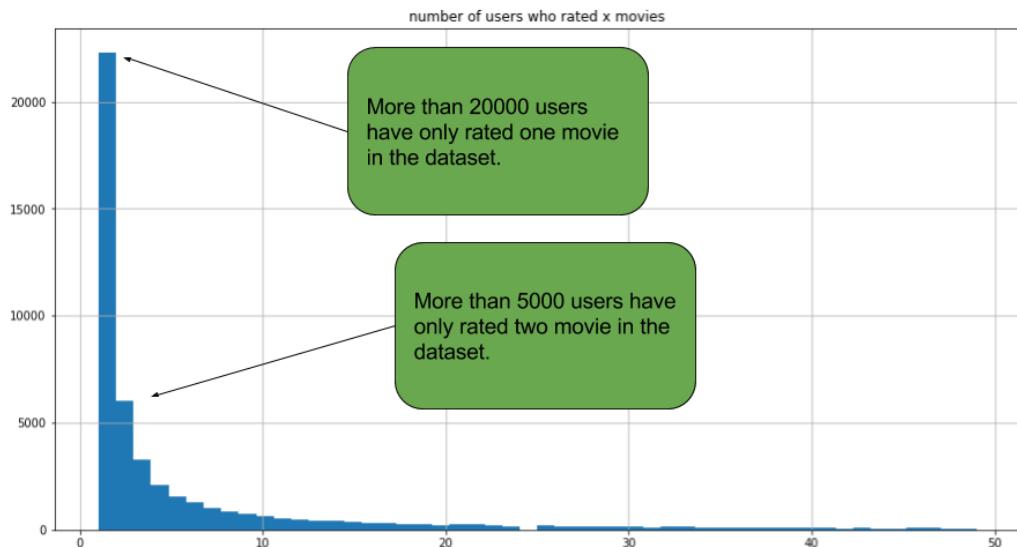


Figure 8.6 Histogram showing the how many users has rated a certain number of movies. For example, you can see that a bit more than 5000 have rated two films (for $x = 2$).

8.5 Ways to select the neighborhood

A neighborhood is simply the set of items that are similar to the content item we are currently looking at. We call it a neighborhood since we talk about items with a small distance between them. There are different ways to define and calculate such a neighborhood. In this section, we will look at three different ways to do that:

CLUSTERING

In the previous chapter, we showed how to implement clustering. If we did user-based filtering we could simply use these clusters as neighborhoods. Or we could rig the clustering algorithm to cluster items instead and use that for the item-based approach. To do the clustering with this purpose we would probably want add more clusters, to not have too big neighborhoods, but it works the same way. The problem with this method is that we might risk that the active user is on the border of a cluster or that the cluster has a strange shape, and therefore we won't get the best neighborhood.

Instead of using the clusters directly as neighborhoods, the clusters can also be used as an optimization to the algorithm, that will narrow the space where the system will look for neighbors. For example, if you have clustered your items into two clusters, the fact that you only have to look at one of them will produce a large performance boost. Of course, by narrowing the space where you search for neighbors, you also risk that they are sub-optimal, so be careful. I would suggest doing test runs with and without clustering and compare if

there is drop in quality (please refer to chapter 9 on how to evaluate recommenders). If you use clustering to narrow the search space, you can use it in cooperation with one of the two following methods.

TOP-N

So the simplest way to find a neighborhood is to say that we define a number N as the number of neighbors that should be in the neighborhood and say that all items have N fellow items in the neighborhood. This will allow the system always to have some items to work with, but they may be items that are not similar at all. This is illustrated in figure 8.7 on the right side. There are two examples showing the results of requiring three neighbors. The first example finds points close to the active point. While the other example needs to go further away. The top- N approach can force the system to use points dissimilar to the active point, making it a bad recommender. For ensuring better quality, you should adopt the next approach instead

THRESHOLD

Another way to cut the cake is by saying that you only want items of a certain standard to be in the neighborhood by requiring that the similarity needs to be above some constant. This is shown in the left of figure 8.7. For the first point, this works great and will get similar items as the Top- N neighborhood. However, you can see some problems with point two since it doesn't have any nearby points. The neighborhood becomes very small and lonely, and potentially empty.

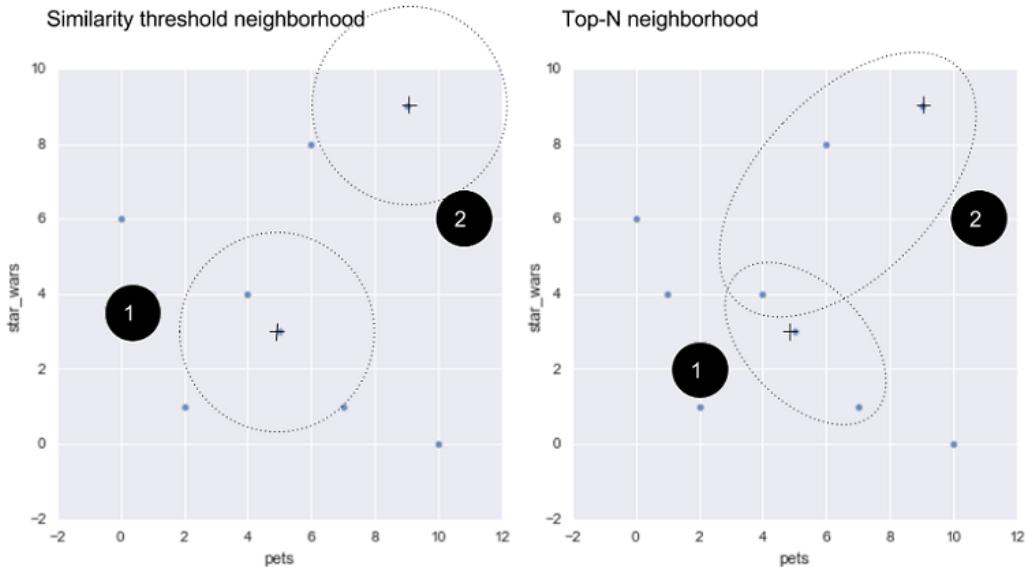


Figure 8.7 Illustrating two different ways of finding the neighborhood. The diagram to the left shows how the threshold neighborhood works. Around the active point, you draw a circle (at least in 2D), and everything that is inside is a neighbor. While on the right is shown the Top-N approach, it doesn't look at distance; it just grows until it has N neighbors in the neighborhood.

Choosing between Top-N and Threshold is choosing between quantity and quality. Choose the threshold method for quality and while Top-N for quantity.

No matter what you choose, you will have the second question to answer: What should the threshold/N be? If you choose the Top-N approach to select the neighborhood, you need to find constant N, indicating the number of points that should be in the neighborhood, but the threshold method requires a threshold constant. The choice depends a lot on both the data and how good you want your recommendations to be. In a utopian dataset, all items will be equally distributed throughout the space, and then the N will be easy to select. But since that doesn't happen very often, it is a matter of balancing between how good the recommendations should be for people liking popular things versus how good recs should be for people with peculiar taste.

8.6 Finding the right neighborhood

Getting back to the example we started in section 8.4, now let's see what happens if we use either Top-N or threshold, using the similarities we calculated in section 8.4, the neighborhoods will look like Table 8.5.

Table 8.5: results of different neighborhood calculations, threshold only returns the elements that

are similar, which the Top-2 mostly returns dissimilar objects also.

	Top-2	Threshold: 0.5
Men in Black (MIB)	ST: 0.63, AV: 1.00	ST: 0.63, AV: 1.00
Star Trek (ST)	MIB: 0.63, AV: 0.35	MIB: 0.63
Ace Ventura (AV)	MIB: 1.00, ST: 0.35	MIB: 1.00
Braveheart (B)	MIB: -0.21, AV: 0.01	
Sense and Sensibility (SS)	B: -0.23, LM: 0.96	LM: 0.96
Les Miserables (LM)	B: -0.32, SS: 0.96	SS: 0.96

As you can see in the table, *Braveheart* will only have other items in the neighborhood, if we use the Top-N method, but if you look at the similarity, it really isn't items which would be similar to it. This means that if we were using the Top-N method, then we would be able to calculate some recommendations based on the N similarities, even if they are not similar at all. While if we used Threshold, we wouldn't be able to recommend anything. With the neighborhoods found we can start calculating predictions.

8.7 Ways to calculate predicted ratings

There are many ways to calculate predicted ratings; but overall, we can split them into two different categories, which I will describe using two different examples (regression and classification, for the machine language savvy!). Figure 8.8 give an overview of the two methods described. When calculating predictions, you will also include the similarity measure in the calculations, which we will also do when we start calculating ratings. One method can be explained using an example of how to predict a sales price for a house, using prices of similar houses. The second method is like collecting votes at a election.

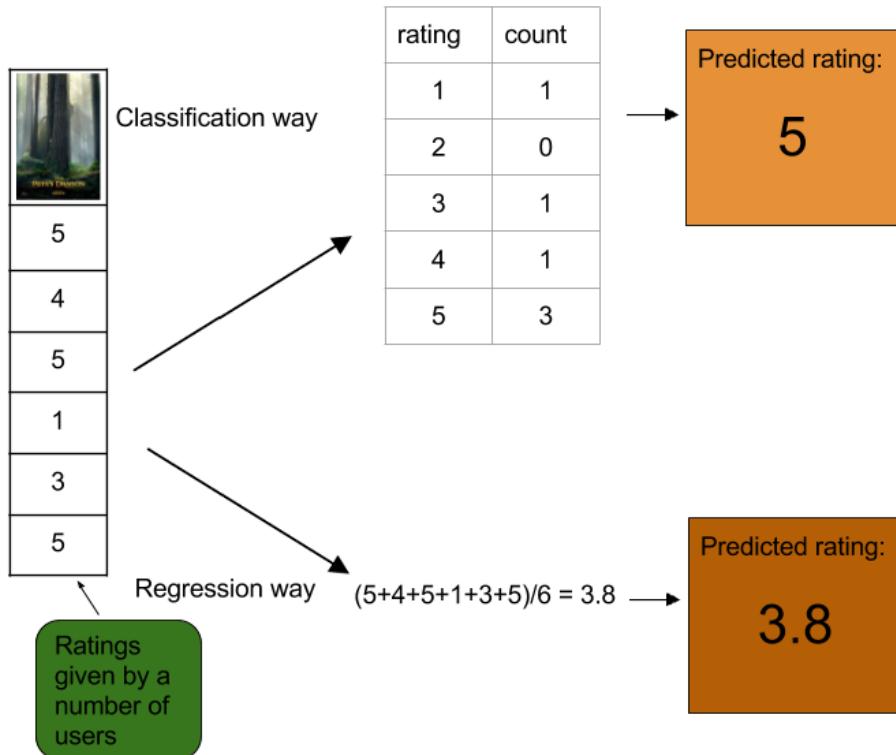


Figure 8.8 Two different ways to calculate predicted rating of the film Pete's Dragon for an active user whose neighborhood has given it the ratings listed to the right. The classification way would produce a rating of 5, simply by seeing which rating has been given more, while the regression would say 3.8, by taking the average of the item-based.

HOUSE PRICE (REGRESSION)

A way to understand how this is implemented is by thinking about the following example of pricing a house for sale.

You want to put your house up for sale, but you don't know what price you should put on it. One way to do it is to find similar houses: maybe you can find similar houses sold in your area and take the average over the sell prices. This method is called regression. Predicting ratings of items for specific users is done the same way. I'm not saying you are a house, but you can find similar users and average their ratings for an item.

FRIENDLY VOTERS (CLASSIFICATION)

Now consider that you have 10 different people who are campaigning to become mayor of your town. You don't know which candidate to vote for, so you ask your neighbors who they

are voting for and count how many neighbors vote for each candidate. When you are finished, you select the candidate with the highest count.

This method can easily be transferred to our domain. You just imagine that instead of candidates for mayor, you have a list of rating values that all candidate as the prediction. If the active user, has five users in his neighborhood, and he wants to know what rating he would give a film. He would first ask how many rated the movie 1 star, then ask about the number of 2 raters, etc. until all rating values have been counted. The active user can now look at the rating count and choose the rating that most of the users in his neighborhood has. Of course, instead of counting each neighbor as one vote you can use the similarity score, and thereby make the votes count only as much as the similarity score

The examples above are simplified because they don't add any weighting to each of the ratings. In the next section, we will look at predicting things in more detail, using the regression technique. Just to recap have a look at figure 8.9. We so far have an active user who has rated (interacted with) a list of items, for each of these items we will find similar items in the neighborhood. These similar items are the candidates for the recommendation.

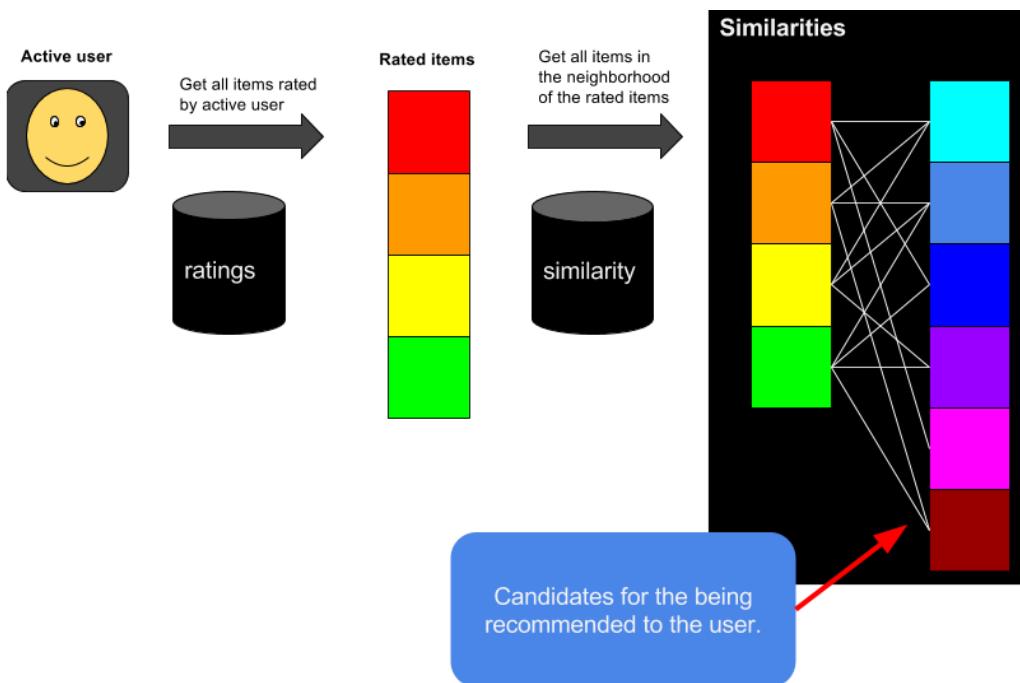


Figure 8.9 The items which candidate to become a recommendation is found by first finding the specific user's rated items and then find the items in their neighborhood.

8.8 Prediction with item-based filtering

Okay, so we are arriving at the core. So, we have created a list of similar items, by looking in the neighborhoods around the users rated items, and now we are ready to do predictions. So how does it work. Exactly? Read on, and it shall all be clear.

In theory, we could take all the items in our catalog and calculate predictions, but since we are basing it on what the active user has already rated, we will just look at items that are in the neighborhood, of these items.

8.8.1 Compute item predictions

Expanding on the regression example from the previous section. We will now look more in detail on how to predict the rating of an item. So given an item that the user liked and its neighborhood we will calculate ratings, we will do a weighted average of ratings of the items that are similar to the item and that the active user has rated. When we do it like this we need to add the users average rating to return the prediction to the same scale as the ratings the user has put in.

To put that into math we write the following formula:

$$Pred(u, i) = \bar{r}_u + \frac{\sum_{j \in S_i} (sim(i, j) * r_{uj})}{\sum_{j \in S_i} sim(i, j)}$$

Where:

1. \bar{r}_u is the average rating of the user u
2. r_{uj} is the active users u rating of item j.
3. S_i is the set items in the neighborhood.
4. $Pred(u, j)$ is the predicted rating for user u of item i.
5. $Sim(i, j)$ is the similarity between item i and item j,

Let's say we want to predict Helle's rating for Star Trek⁴⁸. We will begin by looking at the similarity model we calculated above, and conveniently using the movies where the neighborhood is the same no matter which method we used to find it. So, we have a list like the following:

ST: MIB: 0.63, AV: 0.35

Then it's just a matter of filling in the numbers in the function:

⁴⁸ Yes, it is already there in the rating table, but let's go through the calculation anyway.

$$Pred(Helle, Star Trek) = 2.6 + \frac{0.63 * 0.4 + 0.35 * 0.4}{0.63 + 0.35} = 3$$

If you have calculated along, you should clap yourself on the shoulder because you have just calculated your first item-item prediction.

But wait a minute! That was just a prediction of a rating, how can that turn into a recommendation? That is easy, now you just do this for all items that are of interest and then you order them according to the predicted rating, and return the top 10.

Only if you can find 10 items to predict ratings for, for example if you have implemented a threshold neighborhood then it wouldn't be possible to predict a rating for *Braveheart*. Next problem is that if your algorithm predicts ratings below average, do you then want to recommend them?

Despair not if you are still a bit confused, we will soon look at some code, which even a machine can understand.

8.9 Cold start problems

Collaborative filtering requires data, which is a problem when you receive new users as well as new items – you have no data to use to generate recommendations. Again, a way to get around each of them is to ask new users to rate a few items when they arrive. Present new items obviously to users, as in a new arrivals list. We discussed this in more detail in chapter 6.

8.10 A few words on machine learning terms.

We are going to implement the item collaborative filtering algorithm next. Before that let's get some terms in place.

One of the smart things about item-based filtering is that you can do a lot of the hard work before you have to serve recs to the active users. If you are new to the whole world of machine learning and data science, you might want to know that doing work before hand, is called offline model training. Let's take a look at each of these words:

- Offline: means that you do something before using it live in production. In other words, not while the user is waiting.
- Online: means doing stuff while the user is waiting.
- Model: To create a model means that you take the raw data, run it through your machine learning algorithm, which will aggregate the data and produce a model. Creating a model is usually done offline. Another way to think of a model is that it is a function which you can use to predict ratings (or other things depending on the ML algorithm).
- Training: When you calculate the model you do training. "Training" is a leftover from the AI world where you consider machine learning as a being learning a new skill. In our case, the recommender is training to be able to find similar items quickly.

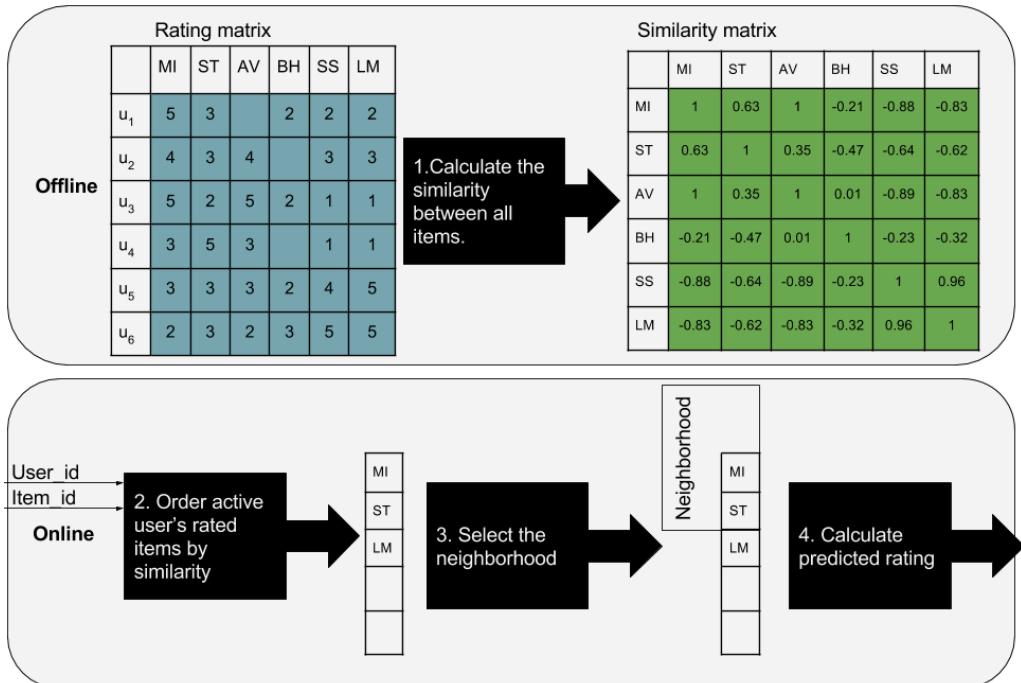


Figure 8.10 Collaborative filtering pipeline divided into offline and online calculation.

Figure 8.10 shows what happens offline and online in item-item collaborative filtering.

8.11 Collaborative filtering on the MovieGEEK site

To get everything into a real-life context, let's take a look at the fictitious MovieGEEKs site. If you haven't already, please download the site from Github and follow the instructions given in the README.md page.

<https://github.com/practical-recommender-systems/moviegeek>

The README.md will also give you details on how to download and import the data.

We will use the recommendations in a second row of recommended items, which, you might remember, lists products that are similar to products a user has already rated or bought. The collaborative filtering recs are shown in figure 8.11. Let's get started building.

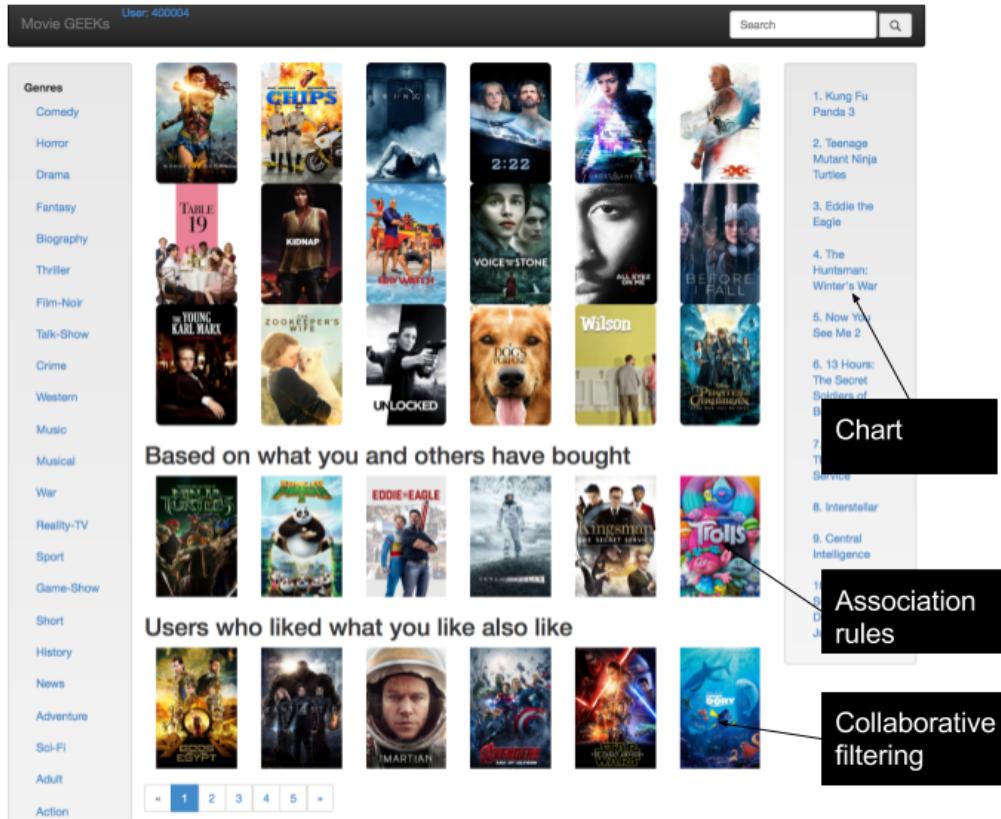


Figure 8.11 Front page of the MovieGEEKs site, including collaborative filtering recs

8.11.1 Item-based filtering

Hopefully by reading the chapter so will have made it crystal clear what we need to do now.

If not then here is a little list of things we will implement. They are also shown in figure 8.10

- Find similar items.
- Calculate predicted ratings for these items
- Use predictions to calculate recommendations

NOTE: The difference between user and item collaborative filtering is how you find the similar items. If you want to read the code then you should look in two places.

- \builder\item_similarity_calculator.py – which is the offline training, ie. the calculation

of all the similarities.

- \recs\neighborhood_based_recommender.py which is the online part of it.

CALCULATING ITEM SIMILARITY OFFLINE

To start out with collaborative filtering we should start building the similarity matrix, which will be a database table with all the similarities calculated. It is better to create a Python script to run outside of the Django scope to do the model building. It might still take hours, but at least it's not days.

We need to create another builder. Overall it will get data from the data base, calculate the similarities, and save them in the database.

Listing 8.1: \builder\item_similarity_calculator.py

```
import os
import pandas as pd

import database
import item_cf_builder

all_ratings = load_all_ratings() ①
ItemSimilarityMatrixBuilder.build(all_ratings) ②
```

- ① Getting the rating data.
- ② Build the similarity model, and save it to the database.

Getting the ratings from the database is really simple, I still encourage you to have a quick look at the code, and if you use it, remember to add a time restriction on the query so you only get the data for some time period. I left it out here, but it is easy to find on Git (method is called `load_all_ratings`).

Let's have a look at the build method which will train/build/calculate/deduce/brew the model.

Listing 8.2: \builder\item_similarity_calculator.py

```
def build(self, ratings, save=True):
    ratings['rating'] = ratings['rating'].astype(float)
    ratings['avg'] =
        ratings.groupby('user_id')['rating']
        .transform(lambda x: normalize(x)) ②
    ratings['user_id'] =
        ratings['user_id'].astype('category') ③
    ratings['movie_id'] =
        ratings['movie_id'].astype('category') ③

    coo = coo_matrix((ratings['avg'].astype(float),
                      (ratings['movie_id'].cat.codes.copy(),
                       ratings['user_id'].cat.codes.copy())))) ④

    overlap_matrix = coo.astype(bool)
                           .astype(int)
```

```

        .dot(coo.transpose().astype(bool)
             .astype(int)) ⑤
    cor = cosine_similarity(coo, dense_output=False) ⑥
    cor = cor.multiply(cor > self.min_sim) ⑦
cor = cor.multiply(overlap_matrix > self.min_overlap) ⑧

movies = dict(enumerate(ratings['movie_id'].cat.categories)) ⑨
if save:
    self.save_similarities(cor, movies) ⑩

return cor, movies

def normalize(x): ⑪
    x = x.astype(float)
    x_sum = x.sum()
    x_num = x.astype(bool).sum()
    x_mean = x_sum / x_num

    if x.std() == 0:
        return 0.0
    return (x - x_mean) / (x.max() - x.min())

```

- ① be sure the ratings are of type Float, otherwise we might have problems later
- ② Normalize the ratings based on the users average, and add it to a column of the data. We use a method called normalize to do it.
- ③ Now convert the user ids as well as the movie ids to categories. This needs to be done to use the Sparse Matrix we use below.
- ④ Convert the ratings into a sparse matrix, called coo_matrix⁴⁹
- ⑤ The overlap matrix is described in more detail below.
- ⑥ Calculate the cosine similarity between the rows.
- ⑦ Removing the similarities which are too small
- ⑧ Removing the similarities not based on big enough overlap.
- ⑨ Create a dictionary of the movie ids, to find the items you are looking.
- ⑩ Save to Db
- ⑪ The normalize method used above.

A word of caution⁵⁰: This code has changed a lot and optimized such that it runs quite fast, so you can run it while you watch, however even if this is fast code, you should expect these things to take a while.

⁴⁹ https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.coo_matrix.html

⁵⁰ In earlier versions of the book this warning was: When you start building this model, run it before going for lunch or something. When I first came up with this version of the code in this chapter (being an optimist I thought I had the final version) waited for 2 days before it was finished. This version leaves me time to go for a run with my dog and eat dinner before it is finished. So, arm yourself with patience; it will take a while. (also take care that your computer has lots of free RAM).

Removing similarities that isn't based on enough overlapping ratings

We have talked about how important it is to have enough overlapping ratings between two items. In many implementations of similarity methods, you don't have the option of define a minimum number of overlapping elements.

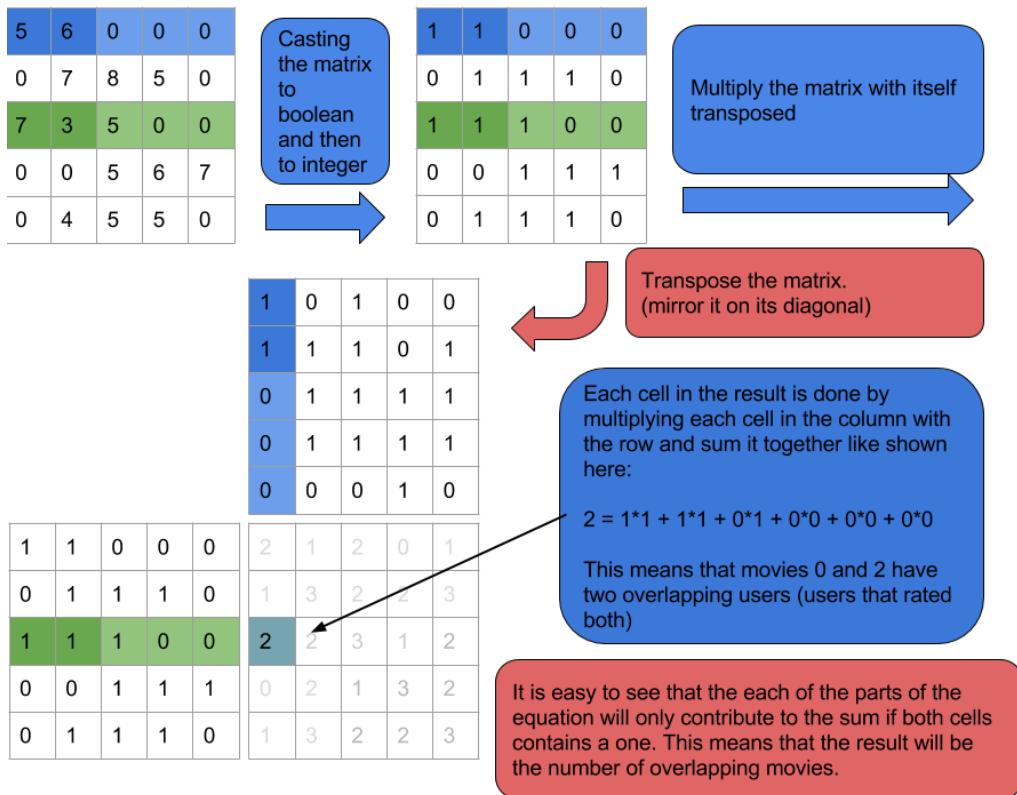


Figure 8.12 The fastest way to calculate overlap on your rating matrix is to do as shown here.

So here is a way that works quite well, at least if sparse matrices. You take your rating matrix, cast it first to Boolean (which makes everything more than zero into True and the rest False) then you cast that to integers which will make True into 1 and False into 0. You can now multiply the rating matrix with a transposed version of itself and presto you have a matrix where you can look up how many overlapping elements you have. This is also illustrated in figure 8.12. It is also done in this line of code:

Listing 8.4: \builder\item_similarity_calculator.py

```
overlap_matrix = coo.astype(bool).astype(int)
    .dot(coo.transpose().astype(bool).astype(int))
```

With overlap matrices, you now have an option where you can say that you want all cells to be True if there are larger than some predefined minimum overlapping. For example, if we only wanted to look at overlaps above 2, then all that is 2 or below is turned to False, the rest True. If we cast that to integer again, then we have a matrix containing ones where we are interested in saving the similarity. This can be done simply by writing

Listing 8.6: \builder\item_similarity_calculator.py

```
overlap_matrix > self.min_overlap
```

This expression will return a new boolean matrix. When we have calculated the similarity, we simply do an elementwise multiplication, which means that you multiply each cell individually, with the cell in the same position. Multiplying the similarity matrix with the Boolean overlap matrix will work such that all similarities that we are not interested in will be multiplied with zero, meaning they will turn zero, while the interesting ones will be multiplied with one, and stay the same. This is done in this following line of code. We do the same trick to remove similarity values that are too small.

Listing 8.5: \builder\item_similarity_calculator.py

```
cor = cor.multiply(cor > self.min_sim) ①
cor = cor.multiply(overlap_matrix > self.min_overlap) ②
```

- ① remove similarities which are too small.
- ② remove similarities which are not based on enough overlapping ratings.

ONLINE PREDICTIONS

To use the model, we will be building with the methods shown above, we need a prediction algorithm. It will do as shown in figure 8.13:

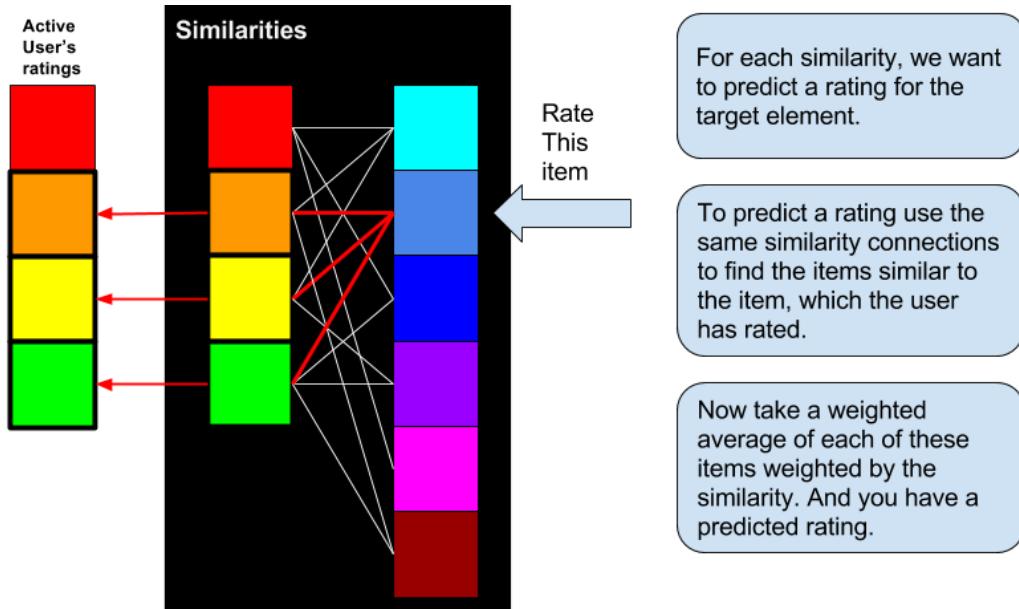


Figure 8.13 To predict a rating, we use the similarity between the item in question and the items rated by the user, and her actual ratings. We multiply each similarity with the users rating, sum that together and divide it by the sum of the similarities. This creates a weighted average.

It is a good point to notice that we only go to the database twice. Once to get the users ratings and ones to get the similarities of these. In fact, an easy way to optimize this would be to join the data in the database before extractions. Such that you would have the similarities as well as the user's actual ratings.

When the similarity script is finished running the offline calculations, it is time look at the online part of the recommender. The following code shows the online part, i.e. the code being run when a user visits the page.

Listing 8.6: \recs\neighborhood_based_recommender.py

```
def recommend_items_by_ratings(self, user_id, active_user_items, num=6):

    movie_ids = {movie['movie_id']: movie['rating']
                 for movie in active_user_items}
    user_mean = Decimal(sum(movie_ids.values())) / Decimal(len(movie_ids)) ① ②

    candidate_items = Similarity.objects.filter(Q(source__in=movie_ids.keys())
                                                &~Q(target__in=movie_ids.keys())) ③
    candidate_items = candidate_items.order_by('-similarity')[num] ④

    recs = dict()
    for candidate in candidate_items:
        target = candidate.target ⑤
```

```

pre = 0
sim_sum = 0

rated_items = [i for i in candidate_items
               if i.target == target][:self.neighborhood_size] ⑥

if len(rated_items) > 0:
    for sim_item in rated_items:
        r = Decimal(movie_ids[sim_item.source]) - user_mean
        pre += sim_item.similarity * r
        sim_sum += sim_item.similarity
    if sim_sum > 0:
        recs[target] = {'prediction': user_mean + pre / sim_sum,
                        'sim_items': [r.source for r in rated_items]} ⑪ ⑫

sorted_items = sorted(recs.items(), key=lambda
                      item: -float(item[1]['prediction']))[:num] ⑬

return sorted_items

```

- ① create a dictionary of movie ids and ratings.
- ② calculate an average
- ③ find all items similar to active users rated items.
- ④ cut the list only to include 20, this is a choice, maybe you should play around with it, to find a good balance between speed and quality.
- ⑤ iterate the similar items.
- ⑥ from the target item, now look at all the similarities where it is present, that gives you a list of the items the active user rated.
- ⑦ if we do have such items (and we need to have otherwise the similarity item wouldn't be there. It is best if it is more than one, otherwise the predicted rating will be the same as the one similar item, also if the similarity indicates that they are very different.
- ⑧ Deduct the user mean from the item rating.
- ⑨ Multiply the normalized rating with the items similarity
- ⑩ Add the similarities
- ⑪ Just to be sure it doesn't add up to zero (and thereby cause a division with zero, which will throw an exception)
- ⑫ Append the predicted rating to the result, along with the rated items.
- ⑬ When we have run through all of them, sort them based on prediction value. Here is another place where you could consider playing around a bit, and add other considerations into the code.

We should now see another list of recommendations. What do you think, do the recommendations look okay? Can you see them in figure 8.11 above?

Opening the site as our persona Helle or look at her analytics page (<http://127.0.0.1:8000/analytics/user/400004/>), we see that the recs contain some somewhat interesting recommendations (what is shown in figure 8.11)⁵¹

⁵¹ To impersonating users you simply add a querystring parameter called user_id. So to see helles page (who has user_id 400004) you request following url: localhost:8000/?user_id=400004

8.12 What is the difference between association rule recs and collaborative recs?

Taking one last look at the recommendations. Comparing the recommendations calculated with the association rules and the ones with collaborative filtering. The association rules look like they are closer to what Helle's taste actually is. But remember we are looking at two different datasets, the association rules were based on the buys of our personas, which is auto generated by the script mentioned in chapter 3, while the collaborative filtering is based on the ratings in the downloaded dataset. In a real system, those would be the same dataset. The reason why associative rules are not collaborative filtering are that they are based on what is in a single shopping basket, not what a user buy over time. While collaborative filtering is looking at what users buy or rate over time.

8.13 Levers to fiddle with for Collaborative filtering

It is not always enough just to implement the algorithm to get good recommendations. Often there are always things that need to be adjusted.

Above we have adjusted the number of overlapping users that is needed before we will calculate a similarity. Let's make a small list of things that can be adjusted:

- Which ratings should be used for the active user.
- Only the positive ones?
- Only the most recent ones?
- How should we normalize the ratings?
- When creating similarity
- How many user's needs have rated two movies for the similarity to be calculated?
- Should we restrict similarities only to be added to the similarity list if they are positive?
- When creating the neighborhood
- What method of selecting the neighborhood should be used?
- How big should the neighborhood be (selecting threshold and/or N)?
- Predicting the ratings
- Should we use classification or regression?
- Should we use a weighted average?
- When returning the recommendations
- Should we return all of them, or just the ones with good predictions, ie. Prediction above some threshold.

And the list goes on and on...

But each of these will narrow down a number of movies coming through the calculations. Take for example the overlap of users that you require. If you look at figure 8.14 you can see how it effects the number of similarities you will have in your model.

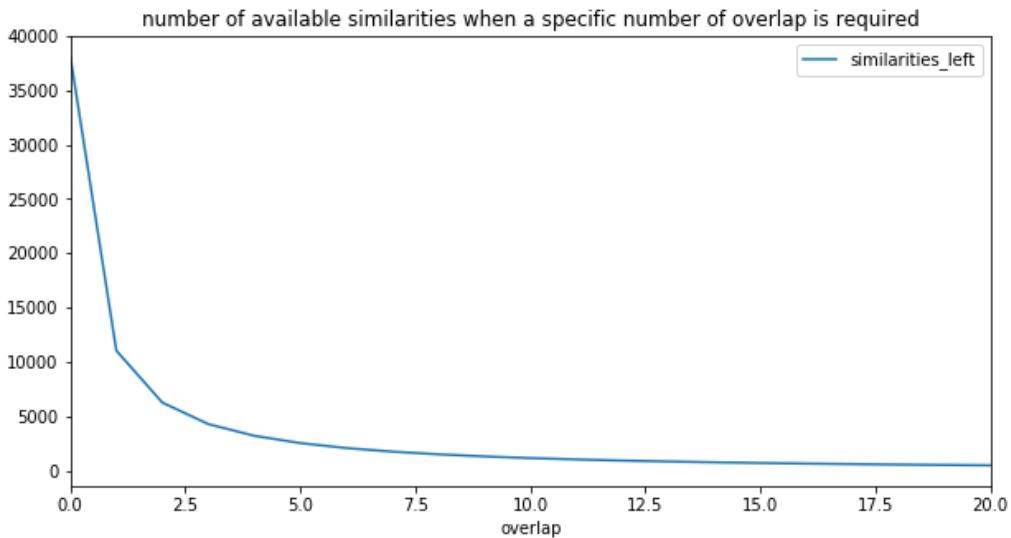


Figure 8.14 plot showing how many similarities you will have when you add a higher required number of overlapping users. Please note that the y-axis is in thousands.

If you don't require any min overlap you have around 40 million similarities, if you require two you are already down to 5 million, and so on. Another thing to consider if you look at figure 8.15 which shows a plot dividing the movies into the number of ratings they have done. So, we have around 11000 movies which has only been rated by one. To calculate the similarity between movies we need more than two users to have rated both movies, which means that we can't use these 11000 items for anything. The number of similarities actually being saved in the database depends on many things. And it is hard to be specific about how restrictive you should be. My advice is to try first not being restrictive at all, see if that works, then try being very restrictive. You will probably arrive to somewhere in the middle.

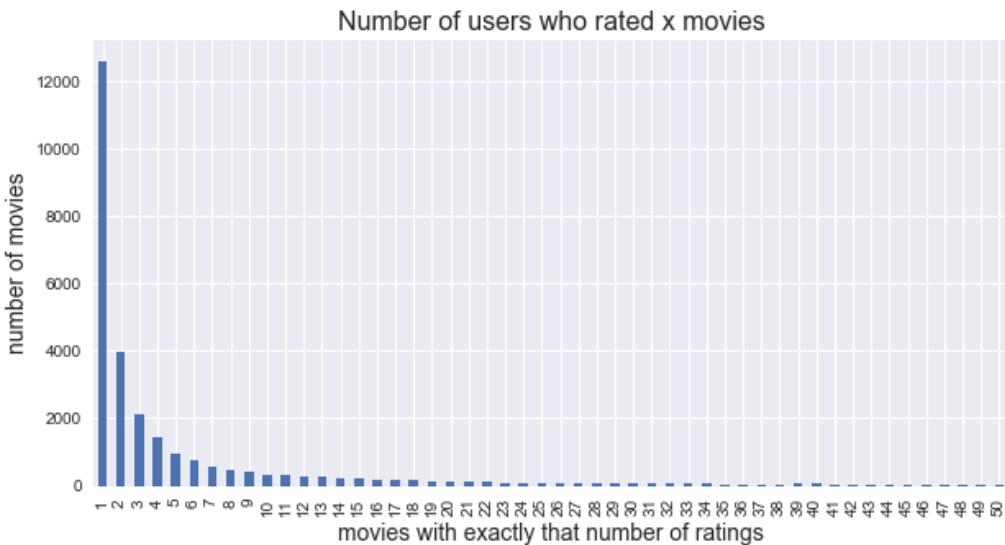


Figure 8.15 illustrates how many movies have received x ratings. For example, if you look at the line above 2 you can read that there are around 4000 movies that has been rated only by two users. If you set a limit of min two overlapping ratings then you can subtract everything to the left of it. So, let's say we require at least 2 ratings then we can see that we already cut away circa 12600 movies. And that is a lot considering we only have 25000 movies to choose from.

Usually, the more users who have rated a film, the more popular it is. That means that as you require more overlapping users, you will also look at more popular items and less personalized movies. The recommendations becomes better, but also, fewer movies will be candidates. Reading through the chapter and indeed the book should have given you an idea of what to choose in each of the options above. In the following chapter, we will look at how to evaluate a recommender, before going crazy about how to select the correct values read that, and then return to this list, keeping in mind what you read in chapter 9. A good reference for what values to start out with can be found in the article "Item-based Collaborative Filtering Recommendations Algorithms" by Badrul Sarwar et al.⁵². As a cliff hanger for chapter 9, have a look at figure 8.16 where some of the evaluation functions are plotted.

⁵² http://files.grouplens.org/papers/www10_sarwar.pdf

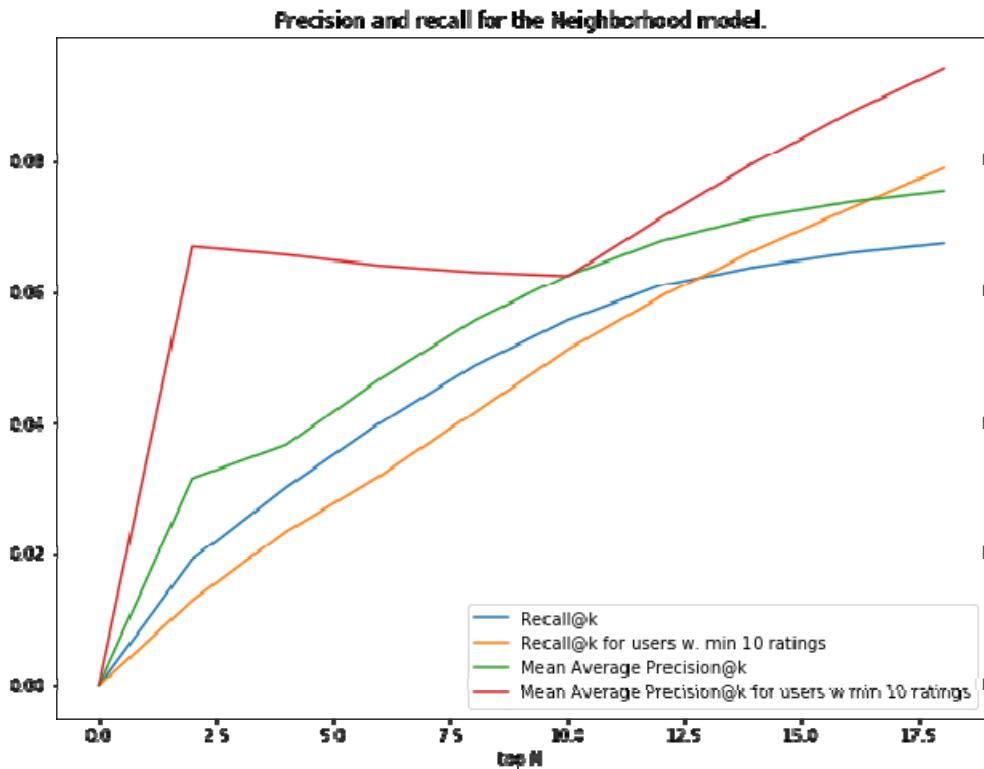


Figure 8.16 A chart showing two of the evaluation measures discussed in chapter 9. Precision is a measure that shows how many relevant elements are present in the recommendation, while recall show how many of the relevant items are shown.

Let's just sum the pros and cons of neighbor collaborative filtering algorithms, before doing a summary.

8.14 Pros and Cons of Collaborative filtering

Even if it sounds fantastic there are some issues and drawbacks that needs to be considered.

- Sparsity problem is one of the biggest, most datasets are not dense enough to recommend other items than the most popular ones. Most users only have a few ratings, and often an e-shop will have thousands of different items. Its can therefore be difficult to find any neighbors.
- Graysheep, which we talked about in chapter 6 is also an issue. Users which have such an extraordinary taste that it is impossible to find related users or items.
- As mentioned many times, to have good collaborative recommendations you need a lot

of ratings from a user before you can trustworthily produce recommendations. This will pose as a problem many places as most systems don't like to sit around while it collects 20 or more ratings before actually starting to recommend things. This problem is usually called the coldstart problem, which was also discussed in chapter 6.

- Collaborative filtering is content agnostic and does not try to fit recs into specific subjects. It therefore follows the behavioral trends of the users. Which will tend to go towards the more popular content. Another way of saying this is that since it is based on similarity, then very popular items will have a lot in common with a lot of content, which will put the popular items in the recommendations often.
- The fact that collaborative filtering is content agnostic is also a great plus. You don't have to spend any energy on adding meta data to your content, or collect knowledge about your users, just the ratings and the interactions between items.

8.15 Summary

In this chapter, we have covered:

- The pipeline of neighborhood filtering can either use user-based filtering, looking at similar users, or using items in item-based filtering.
- Use user-based filtering if there are many more items than users, otherwise use item-based filtering.
- A similarity matrix makes it possible to quickly look up similar items.
- Using the similarity table enables the system to make neighborhoods using either clustering, Top-N, and Threshold methods.
- The neighborhoods we find allow us to calculate predictions as we have a small set of similar users.
- Amazons first stab at recommender system was item-based collaborative filtering.

In the following chapter, we will look at how to evaluate recommender systems.

9

Evaluating and testing your recommender

The Netflix Prize abstracted the recommendation problem to a simplified proxy of accurately predicting ratings. It is now clear that this is just one of many components in an effective industrial recommendation system. They also need to account for factors like diversity, context, evidence, freshness, and novelty.

Xavier Amatriain et. al.⁵³

Evaluating this chapter, you should come up with the following things covered:

- You will learn how to evaluate the effectiveness of a recommender algorithm.
- Before evaluating a recommender system, you will need to understand how to split datasets into training and test data
- Offline experiments are needed to evaluate recommender systems, You learn how to do these and how to evaluate the results.
- Online testing is also needed, you will also have a brief tour of how to go about that.

9.1 Business wants lift, cross-sales, up-sales, and conversions

Why did you implement a recommender? What did you want to gain? Do you want to earn more? Have more visitors? Or simply try out some new technology? No matter what you answer, it might not directly translate into a way to calculate whether or not you are

⁵³ Xavier Amatriain et. Al. Past, Present, and Future of Recommender Systems: An Industry Perspective, Recsys '16

improving. You often hear about algorithms that are better or improve slightly to the current cutting-edge, but improving what and how?

This chapter is about evaluating recommender systems, or rather how to attempt to do it. There is a general agreement among recommender system researchers that it is close to impossible to evaluate a recommender system or algorithm without having a live system to test it out. However, it is very important to be able to tell if your recommender system is going in the right direction. We will go through how best to evaluate a system using the data we have. We will also talk about how to make live tests in a live website, and look at how we might implement it in MovieGEEK, but while we can simulate visitors and visits to make the algorithms have data to run on, it really doesn't make sense to simulate visits for evaluating recommenders.

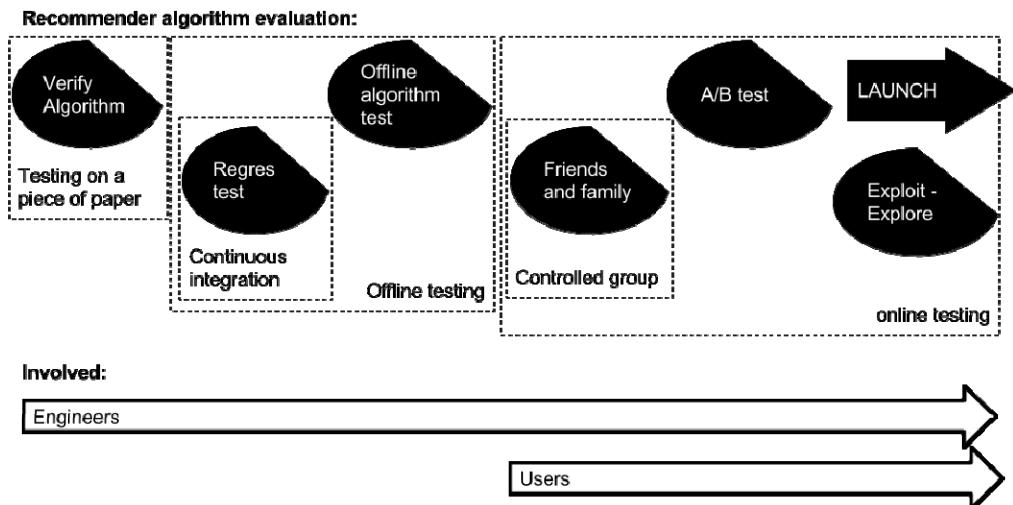


Figure 9.121 The evaluation cycle of recommender systems and who is involved in the process.

Figure 9.1 illustrates the evaluation cycle of a recommender system; we will look closer at each of these. Some of the tasks aren't usually associated with evaluation of recommender systems but they are important if you want to develop and maintain it.

One thing that is often forgotten is that all data applications are living things (not in the AI sense) and will require maintenance and monitoring. Performance is a product of the data being used, which is updated constantly. This means that the behaviour of the system will change if the data does and the system's performance and predictive power might diverge or degrade,

Let's quickly run through the tasks shown in Figure 9.1 before diving into the details. The first step is to verify the algorithm. Take a small dataset with something like ten users and five items and see if the algorithm makes sense. When that is satisfied, then the algorithm can be implemented. It's a good idea to have regression tests continuously running nightly in such a way that if somebody makes an update it won't hurt the performance of the system. When the algorithm is coded and functionality is tested (continuously), it is time to do offline algorithm evaluation using some dataset. If the offline evaluation is successful, you can introduce the output to humans.

Often you will start out with a controlled group, which many call *friends and family*. This is where you will have to answer to your boss (who didn't read this book and doesn't know anything about recommender systems, but is an expert and knows there is something wrong with his recommendations because they didn't look perfect to him). *Family and friends* will only get you so far and to ensure everything is great with the recommender system, it should be unleashed upon the happy users – but just a small percentage to verify it works and increases the key performance indicators (KPI) that you are after. If that is successful, it's time to launch the algorithm for everybody. Many will probably stop there, but a trend that is gaining attention is Exploit/Explore, which we will look at briefly in this chapter.

There are a lot of things to talk about, so let's dig in. Before we start talking about how to evaluate, let's talk a little more about why to do it.

9.2 Why is it important to evaluate?

Is it running correctly?

After having sweated, bled, and sworn to make a recommender system work, you don't want it to fail. And often testing a recommender system is a quest for finding proof that it works. You also hope *not* to find signs that it might not work perfectly or that it returns strange results to more than half of the users. It would certainly be good to know this was going to happen before all your users did.

Often a system will be over-fitted to the taste of the stakeholders... the boss calls and says *my daughter didn't like any of the recommendations your system produced. Fix it or you don't have to fix anything else* and he says it on a Friday afternoon. Maybe its stretching it a bit, but you need to understand who the customer of your evaluation is – because different people want different reassurances.

With that in mind, it is important to have a clear idea of what question you will actually be asking. Let's talk about Netflix again. Their goal is retention of subscribers, but that is only measurable once a month, so Netflix has deduced (probably by looking at their data) that retention of a user is correlated with the user watching a lot of content. But a user watching a lot of content can also indicate other things, not necessarily that they'll be around a long time. Maybe the user's girlfriend just left him, or something else entirely. Getting back to the topic

at hand, though, the goal might not be immediately measurable, and the things that are measurable might not uniquely indicate what you want to know.

Since we are only interested in the actual recommender system, we are interested in an evaluation that shows which of the recommenders give the best result based on the MovieTweetings⁵⁴ data we are using.

To evaluate something that happens to a lot of people (in our case, what “happens” is that a person receives a recommendation) we need to dive into statistics. I am not going to try to teach statistics in this book, but will give you an overview of what you should consider when testing the recommenders, you have implemented. First, we’ll frame our question as a hypothesis.

HYPOTHESIS

The hypothesis describes the goal of the test. For us, that could be the following:

Recommender B will produce recommendations that are clicked on more often than recommendations from Recommender A.

“Clicked more often” is also called the click-through rate or simply CTR. Is this clear enough to run a test? If you show this to the mailman, will he understand it in exactly the same way as you do?

⁵⁴ <https://github.com/sidooms/MovieTweetings>

9.3 How to interpret user behavior.

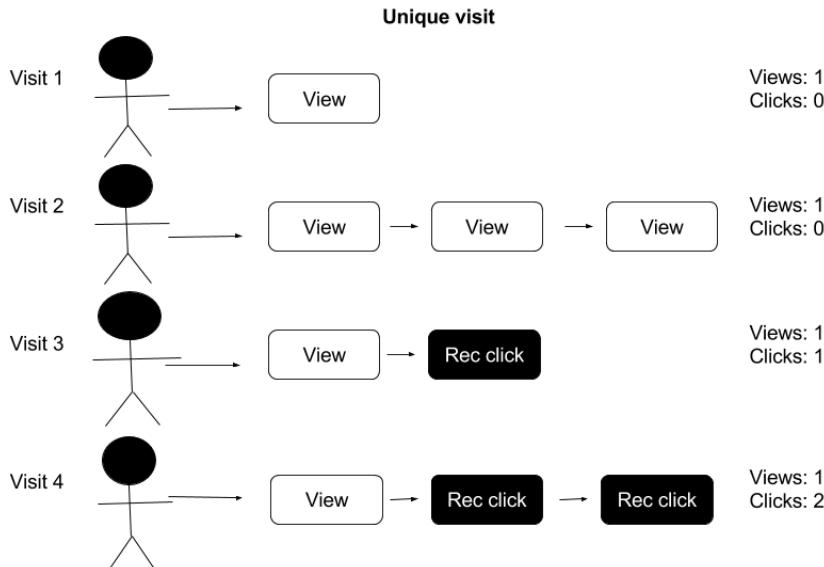


Figure 9.222 different scenarios of user behavior.

Figure 9.2 shows four different scenarios of a visitor arriving on a website. The first visit is quite simple: your system records that it has shown the landing page, and then nothing happens. Should we then assume that the visitor has seen the recs, or not? Visit 2 shows a visit where several different pages are viewed; here we must assume that the recs are presented but the visitor didn't show any interest, since he clicked on something different. Visit 3 is perfect, visitor arrives and clicks on a recommendation. How should we value Visit 4, though? Does it count as one click or two? Possibly it would make more sense just to say that a recommendation algorithm is successful if a user clicks one or more times on the recommendations. But again, that depends on the domain you are in.

9.4 What to measure

I think there are two types of goals, one is that we are here to make our customers happy and hopefully earn some money at the same time. The other is we want to make as much money as possible, and if we must make our customers happy to do that, then we will do it, but just enough to earn money.

You, the developer, need to make your boss happy, which she will be if the customers are happy (and spending money). Customers' goals are nearly the same – they want to be happy

and will spend money if they are, but they don't have your boss's goal of maximizing their spending!

Let's forget about the money for a moment and think about the happy customers. How can we make the customers happy? The answer to this question is very individual and differs from domain to domain, but here is what makes me happy when I shop online:

1. Understand my tastes, I do not like to receive recommendations that are obviously wrong.
2. Give me a nice variety of recommendations, I want my recommendations like I want nature – full of diversity.
3. Surprise me, I want to see things that I never knew I wanted, but now I want it.

As a site or content maintainer, you'd also want to add:

4. Cover all the catalogue.

We will look a bit more at the details on each of these.

9.4.1 Understanding my taste – minimizing prediction error

One way of measuring how well a recommender understands my tastes is by saying that it must predict whether I like an item that I know and have already rated. This can be done by measuring how often the recommender gets close to predicting the correct rating. Another way to look at it is as decision-support metrics, which look at dividing the predictions into groups where the user will react in similar way. If the system predicts a film above 7 on a 1-10 scale, then I would probably be interested in watching it, below 7 but above 3 would be a film that I wouldn't refuse to watch if my wife wanted to watch it. While below that, I would put my foot down and say "No way!" If the recommender predicts a rating that is somewhere inside the ranges that fit where I would predict it, then it is probably ok. If it is in a different group then it will make me lose time and/or lose good content. And eventually lose confidence in the recommender if it happens too often.

9.4.2 Diversity

Let's take a quick detour into The Bible, where we find this in the book of Matthew:

For unto everyone that hath shall be given, and he shall have abundance: but from him that hath not shall be taken even that which he hath.

— Matthew 25:29, King James Version.

You are probably asking how this relates to recommender systems. Well, this quote is the basis of what is called the Matthew Effect. You might be more familiar with it something like *the rich become richer while the poor become poorer*. If we view a popular item as rich, and unpopular is poor, then the issue here is that we will have items in our catalogue that are recommended often because they are popular. Of course, it is good that popular items are

recommended often, but these popular items might show up in too many recommendations and therefore become even more popular, while other potentially great items don't get shown simply because they have not been popular yet. With popular items always being favored, we are also creating what we call a filter bubble⁵⁵, which is good because you will always get recommended what you like. But it also means that you will never know if there is something slightly different which you like even more, but just haven't experienced it yet. Besides the filter problem there is also the fact that the original idea of a recommender system is to help users navigate a larger catalogue than what you have in a shop, if the recommender only shows popular items, then that advantage is lost (okay I know there are lots of other advantages as well, but this is an important one)

It is, however, very hard to measure whether or not your system is successfully diverse and also quite hard for your system to be diverse when it's personalized. Researchers have attempted to calculate a diversity measure by calculating the average dissimilarity between all pairs of the recommended items. Have a look at "Improving Recommendation Diversity" by Bradley et. al. for more on the subject.

9.4.3 Coverage

Diversity as we talked about above, leads nicely into coverage, because the better the diversity the better the coverage. One of the main reasons for implementing a recommender system is to enable users to navigate the full catalogue, which is known as content coverage. Coverage refers both to ensuring the algorithm will recommend everything in your catalogue and whether it can recommend something to all users registered, the last is called user coverage.

The brute force way of calculating the user coverage is simply to iterate over all users, call the recommender algorithm, and then see if it returns anything. This could be done with the following code:

Listing 9.1: evaluator\coverage.py

```
def calculate_coverage(self):
    for user in self.all_users:
        user_id = str(user['user_id'])                                1
        recset = self.recommender.recommend_items(user_id)            2
        if recset:
            self.users_with_recs.append(user)                         3
            for rec in recset:
                self.items_in_rec[rec[0]] += 1                           4
                5
                6

    no_movies = Movie.objects.all().count()
    no_movies_in_rec = len(self.items_in_rec.items())
```

⁵⁵ https://en.wikipedia.org/wiki/Filter_bubble

```

no_users = self.all_users.count()
no_users_in_rec = len(self.users_with_recs)
user_coverage = float(no_users_in_rec / no_users)
movie_coverage = float(no_movies_in_rec / no_movies)
return user_coverage, movie_coverage

```

7

8

- 1 run through all users.
- 2 get recommendations for the user
- 3 if the recommender returned recommendations
- 4 append user to the list of users that received recs
- 5 for each recommended item
- 6 add to items in rec.
- 7 Calculate user coverage
- 8 Calculate item coverage

This method solves both coverage calculations mentioned above. User coverage is defined as follows:

$$\text{coverage}_{\text{user}} = \frac{\sum_{u \in U} \rho_u}{|U|}$$

Where

$$\rho_u = \begin{cases} 1 & : \text{if } |\text{recset}| > 0 \\ 0 & : \text{else} \end{cases}$$

Using the result of the method execution we can also calculate the catalogue coverage, which is defined by:

$$\text{coverage}_{\text{catalogue}} = \frac{|\text{all items in recs}|}{|I|}$$

where

$$|I| = \text{number of items in the catalogue}$$

For this we will also need the total number of items in the catalogue.

Using the code `calculate_coverage` method from Code listing 9.1. I found that the item-item collaborative filtering method implemented in chapter 8 has a coverage of 13%. This means that out of the 26,380 movie items in the dataset, only 3,473 movies would actually make it into a recommendation⁵⁶. While running this code, I also collected which movies were actually recommended. This resulted in the histogram seen in Figure 9.3, which shows how many movies were only recommended once between all the recommendations produced for all our users. The movies that are really popular are the ones that are in the long tail (which is a bit

⁵⁶ This of course depends on what parameters you use to train it.

counter intuitive compared to the long tail problem we talked about before). The figure was actually cut because there are movies that have appeared in more than 100 recommendation sets.

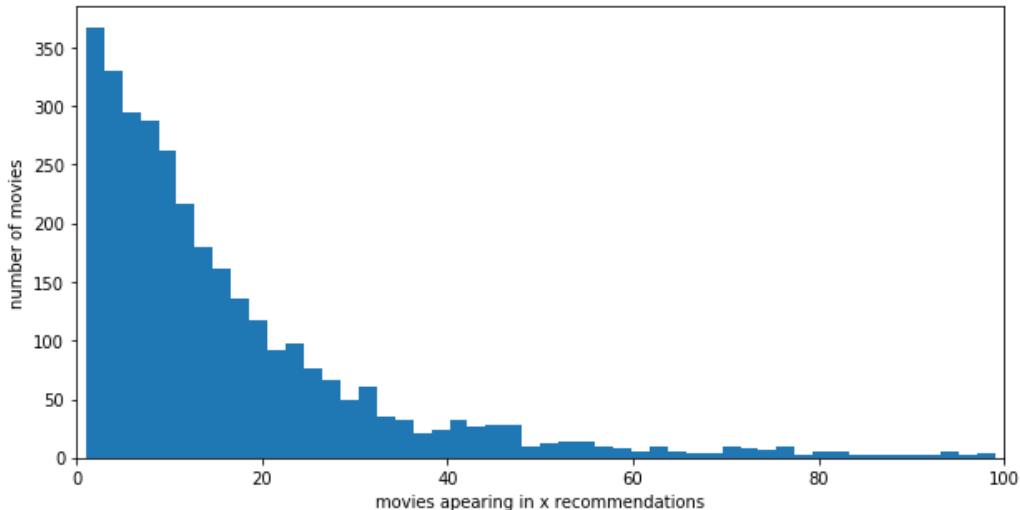


Figure 9.3 shows how many movies are shown x number of times. More than 350 movies are shown in only one recommendation. Counter intuitively, the movies that are really popular are the ones that are in the long tail.

3473 movies out of 26380 is not a lot. So maybe it is a good idea to add a content based recommender also which we will talk about in the following chapter.

The coverage code shown above can be run by running

Listing 9.2: To run the coverage for the collaborative filtering algorithm

```
>python -m evaluator.coverage -cf
```

if you run the `--help` instead you will see that you have an option to run the coverage evaluation on each of the algorithms we will look at in the rest of the book.

Listing 9.2: To run the coverage for the collaborative filtering algorithm

Evaluate coverage of the recommender algorithms.

optional arguments:

- h, --help show this help message and exit
- fwls run evaluation on fwls rec
- funk run evaluation on funk rec
- cf run evaluation on cf rec
- cb run evaluation on cb rec
- ltr run evaluation on rank rec

But please wait with running the others until you have trained models. Otherwise it will take a real long time to calculate coverage.

9.4.4 Serendipity

We want to be surprised by finding things in our recommendations that we love but never knew we would. Serendipity is about giving the user that sensation so that all of their visits aren't just more of the same.

Serendipity is very subjective and difficult to calculate, so I will just ask you to remember that it's important, and make sure that you don't constrain the recommender's returns too much. As constraining means less serendipity.

You can read about attempts to apply metrics Serendipity in the article *Beyond Accuracy: Evaluating Recommender Systems by Coverage and Serendipity*⁵⁷

We now have some different concepts of what we should measure and evaluate. We are one step closer to doing evaluation, however, before getting there we need to look at some other things that I need to get off my chest as a former software developer.

9.5 Even before implementing the recommender

The lamp will only burn as good as the oil you pour on it.

There are a few steps to consider:

- Verify the algorithm
- Verify the Data
- Do regression test

Let's look at these in more detail.

9.5.1 Verify the algorithm

It's a silly thing to add, but you'd be surprised how often somebody reads a scientific article that sounds so cool, then months are spent implementing it before somebody finally thinks to ask about the elephant in the room. Too late they realize that this algorithm won't work with the data available or produce the output they need. Do yourself a favor and write down a simple scenario in which you rigorously calculate your way through a small example where the algorithm is used. Consider carefully what data you can provide as input, and be sure to agree with stakeholders what is going to be the output.

This simple scenario can also work as a way to verify that the system is running. The algorithm you will look at in Chapter 9 takes a couple of hours to run on the full dataset. If

⁵⁷ <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.464.8494&rep=rep1&type=pdf>

you are using that to debug the algorithm you will never finish. You will do one correction and start of the builder, then life happens and when it is finished you will have forgotten everything and if really unlucky next step you do is to do a new correction which basically undo the correction you did first.

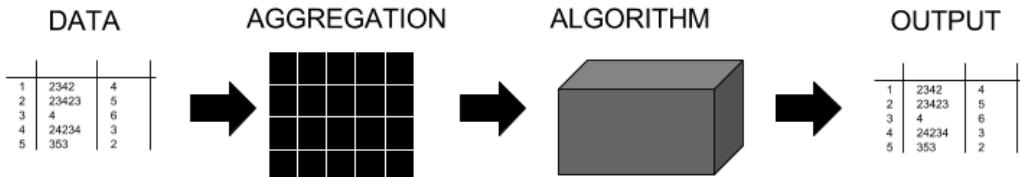


Figure 9.4 Execute the algorithm by hand to test if you have data and if it produces the right output

DATA

Test if the data you need is either available or possible to produce. Is the data persisted far enough into the past to be usable? It's great if you save all user's ratings and behavior, but if the system deletes everything that is a day old, you might have a problem. You should also look at the diversity of the data, do you have data of users who interact with all your catalogue or only some of it, in which case you need to do something special to get those untouched items into play.

ARCHITECTURE AND AGGREGATION

Having the data is not enough. Can you retrieve it? If so, can you do the aggregation that you want? In SQL databases, it's easy to do joins between tables, but that might not be the case in a database like MongoDb. It seems obvious, but remember to consider this before starting to rely on the data.

ALGORITHM

How complicated is the algorithm? Is it easy to implement? Does it require some mathematics that are difficult to implement or performance wise too much to do on a computer? And if you create a small example using a small data set, is it possible to make a naïve implementation in, for example, a Jupyter notebook to illustrate that it can be done?

When these things are satisfied, you can start implementing your algorithm.

9.5.2 Regression Testing

As a software engineer you should know about regression testing, which means that you should have some test set that can be run, either nightly or at least every time somebody makes a change to the code base.

Often people will argue that these algorithms are way too complicated to run automatic tests with. If you run into one of those people, please rap them over the fingers, especially if they claim that the business can't afford it.

Take for example the collaborative filtering algorithm, it is built as a pipeline with several steps, if we look at each, then we can test the pieces, for example the similarity method can easily be tested if it responds correctly to simple vectors. If you call the similarity method with the same vector the similarity should return one, while two vectors that are orthogonal on each other should be -1 or 0 depending on which similarity function you use. Have a look at the tests I have done in the test folder of the project⁵⁸. One of the similarity function tests looks like the following listing.

Listing 9.3: test\item_similarity_calculator_test.py

```
def test_simple_similarity(self):
    builder = ItemSimilarityMatrixBuilder(0)

    no_items = len(set(self.ratings['movie_id']))
    cor = builder.build(ratings=self.ratings, save=False)
    self.assertIsNotNone(cor)
    self.assertEqual(cor.shape[0], no_items, "Expected correlations matrix to have a row for each item")
    self.assertEqual(cor.shape[1], no_items, "Expected correlations matrix to have a column for each item")

    self.assertAlmostEqual(cor[WONDER_WOMAN][AVENGERS], -1, "Expected Wolverine and Star Wars to have similarity 0.5")
    self.assertAlmostEqual(cor[AVENGERS][AVENGERS], 1, "Expected items to be similar to themselves similarity 1")
```

I have created a small dataset to test on, which enables me to run small tests to verify that step of the collaborative filtering pipeline is working.

When this is in place it is time to start looking at the offline evaluation, which is typically where we start talking about evaluation of machine learning and recommender algorithms.

9.6 Types of evaluation

There are several ways to test a recommender algorithm and not all of them are going to give you an accurate view of how the algorithm will perform if you add it to your website. The sad truth is that the only way to really know is to put it in action on your site. But before we do that we can ease the transition a bit by doing some other evaluations first.

We will assume that you have a data set containing ratings already. To do a true evaluation we need what is called a complete ground truth set, which is a dataset containing

⁵⁸ <https://github.com/practical-recommender-systems/moviegeek/tree/master/test>

information about all combinations of users and content. If we had that, then maybe we don't really need a recommender system, as the user already would know exactly how it feels about all items. Since we don't have that, we need to instead to assume that the user item combinations that are present in the data is the truth and representative.

There are three types of scenarios you can test in – offline experiments, controlled user experiments, and online experiments. There is a lot of talk among recommender system researchers, look for example at the article “Accurate is not always good” by Sean McNeer⁵⁹ about offline experiments not working and controlled and online experiments being too expensive, but sometimes it's just hard to make everybody happy. Each of these has a purpose; you just need to select the right one for the job.

9.7 Offline evaluation

The idea of offline evaluation is that we use data that we regard as truthful. Split the data into two parts and feed one part to the recommender. Use the other part to either verify that the recommender predicts ratings on items in the set that was hidden to it, that are close to the actual ratings or produce recommendations that contain items that were highly rated in hidden data, as shown in figure 9.4. Or we can say that items a user rated, he consumed, even if he didn't like them, so they still count for something.

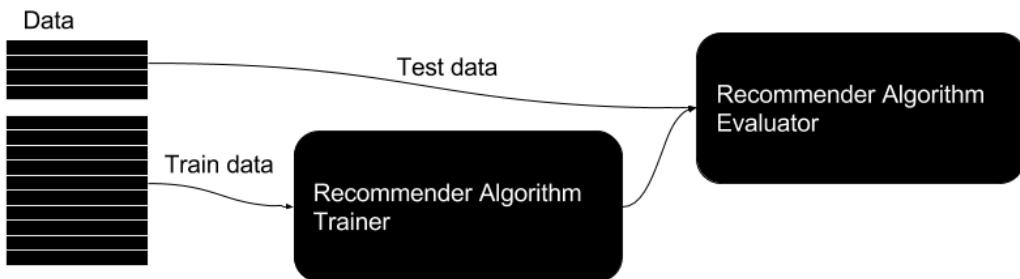


Figure 9.5 Offline Evaluation usually splits data into a training and test set, before testing the recommender algorithm

This is not regarded as a good way of evaluating recommenders, but so far it has been hard to come up with a better way - unless you are Netflix and you can do live A/B testing on new features (which we will talk about in a bit). Offline evaluation has continued to be a way to measure the effectiveness of the recommender. To complicate matters further, a recommender can pass an offline evaluation with flying colours but still fail miserably in

⁵⁹ Accurate is not always good: How Accuracy Metrics have hurt Recommender Systems <http://www-users.cs.umn.edu/~mcnee/mcnee-chi06-acc.pdf>

production. Of course, that is a lot of ifs and maybes, but we will try to do this with the recommender we looked at in chapter 8 and in the following chapters we will talk about how to evaluate the algorithms we look at. If you are looking into doing research into recommender systems its good if you have a good understanding on how to implement, test and present new algorithms, or dialects of existing ones. I recommend looking at Michael Ekstrands et al's article on the Recommender Research Ecosystem⁶⁰

9.7.1 What to do when the algorithm doesn't produce any recommendations

Often you will run into cases where the recommender doesn't produce any or maybe not a long enough list of recommendations. In case of the evaluation it can be a problem, so often you fill in the blanks with some simple algorithm, like most popular items, the average rating for the item, or the average of the user ratings. A slightly more complicated solution is something that is called baseline recommenders, which we will look at in chapter 11. This is also shortly discussed in the article I recommended above.

9.8 Offline experiments

An offline experiment uses the data that you have and measures whether the algorithm is good or not. There are some things to consider when making an offline experiment.

With offline experiments, you have limited options for finding out what you really want to know because the data you have will probably be based on behavior of users in a scenario where there was either no recommender system or there was one that you were hoping to show is inferior to the one you want to test. This means that you can only test if it is as good (or as bad) as the one used to collect it.

We have agreed that a great power of a recommender system is to provide users a selection of surprising new items that they would like to use. But, if we only have data where the level of surprising items is not so high, then how can we test it? We would need users to provide feedback on the complete catalogue to be able to know what a good recommendation is.

For now, since we don't have any other ways of testing our algorithms before exposing humans to it, it will have to do.

How will we measure what is good?

MEASURING ERROR METRICS

We will measure the difference between the ratings we have in our historical data with what the recommender algorithm outputs. So, when we talk about the error it is exactly that. The difference between a user's rating and the recommender's prediction of that rating.

⁶⁰ Rethinking the Recommender Research Ecosystem: Reproducibility, Openness, and LensKit - <http://files.grouplens.org/papers/p133-ekstrand.pdf>

$$\text{error} = r - p$$

In chapter 7, we looked at mean absolute error (MAE), mean squared error (MSE), and root mean squared error (RMSE). To quickly summarize, the MAE takes the absolute value of each of the differences and finds the average of those. The reason for taking the absolute value is that if the recommender guesses low in one prediction and high in the next one, then those two would cancel out each other, but if you remove the sign then there are all positives. And this is what we want so that we can measure the distance between the two.

	Actual	Predicted	Error
	★★★★★	★★★★★	★
	★★★	★★★	★
	★★★★★	★★★★	★
	★★★★	★★★	★
	★★	★★★★★	★★★★★

Figure 9.6 Users should have equal weight in the evaluation. Two user's ratings are shown, one has added four ratings, while user two has only added one. If you do an average over all ratings then user 2s bad experience will drown in user 1s good predictions.

What they all have in common is that they sum up all errors (squared or not), if there are users who has a lot of ratings the recommender would probably know better how to predict the ratings, while users with only few would be bad. The RMSE will penalize big errors a lot, such that one big error will count a lot more than several smaller ones. While if you use the MAE big errors or outliers doesn't push the error so much. if it is important that none of your users gets bad recommendations then you should use RMSE. But if you realize that you can make all users happy, then it is probably enough to use MAE.

In figure 9.6 we have illustrated two users, the top user has added four ratings and let's say the system knows him well, so it is good at predicting ratings for him, while the bottom user has only added one rating, so the system doesn't know him too well. If we average over all the prediction errors, we will get $(1+1+1+1+3)/5 = 1.4$, while if we average the users' errors first then we get $((1+1+1+1)/4 + 3/1)$. We have an average of 2 as error, which better indicates the overall user experience.

On the hand if you look at each item instead, the popular items will be the ones easier to predict, so if you have a lot of items in the long tail, it might be worth thinking about what you are optimizing for. Maybe you can remove the popular items from the test set. Or you can divide the data (if you have enough) into a set with popular items, and evaluate that, and another dataset with long tail items which you can evaluate separately.

If we just took the average of all the rating errors, the majority of our result would come for the user we knew a lot about, while the one with the few ratings didn't contribute a lot. Get around that by measuring for each user and then take the average over those. Then each user would contribute equally and the evaluation wouldn't favor users with many ratings. On the other hand, if you are just out to verify the predictive error on all the test data then you should average over all of them. Like this:

$$MAE = \frac{1}{|RECSET|} \sum_{r \in RECSET} |r - p|$$

where

RECSET= set of all recommended items for all users

As mentioned a few times throughout this book, it might not be too important to the users if the recommender can predict their ratings down to a decimal(Unless you are trying to win a competition. It's likely they would like it to just list the things they like and leave out the things they don't like. (By the way most people don't actually know how to describe what they do and don't like, so asking the recommender to do it is already quite a complex requirement)

Using the metrics, we talked about here presents another issue, which is that all ratings are considered equally important. If we look at the problem from a top-N recommendation instead of rating prediction, then we would be very interested in having good recommendations in the top but not care so much about what happened further down. Let's look at some rank aware metrics.

MEASURING DECISION-SUPPORT METRICS

Decision support is about taking each element at say was the system right or wrong. So if we consider a recommender system, and look at each recommended item, an compare it with a user's actual consumption we can have four outcomes. For a given item, it can either be recommended or not, and the user can either have consumed it or not. If the recommender recommends the item, we say its positive, and then if the user had consumed the item, we say it was the right decision, ie. True positive (TP), if the user didn't then it is False positive (FP). If the recommender didn't include the item in a recommendation and the user consumed it, it was a false negative (FN), while if the user didn't consume it it's a true negative (TN). Usually it is depicted as table, which is shown in figure 9.7

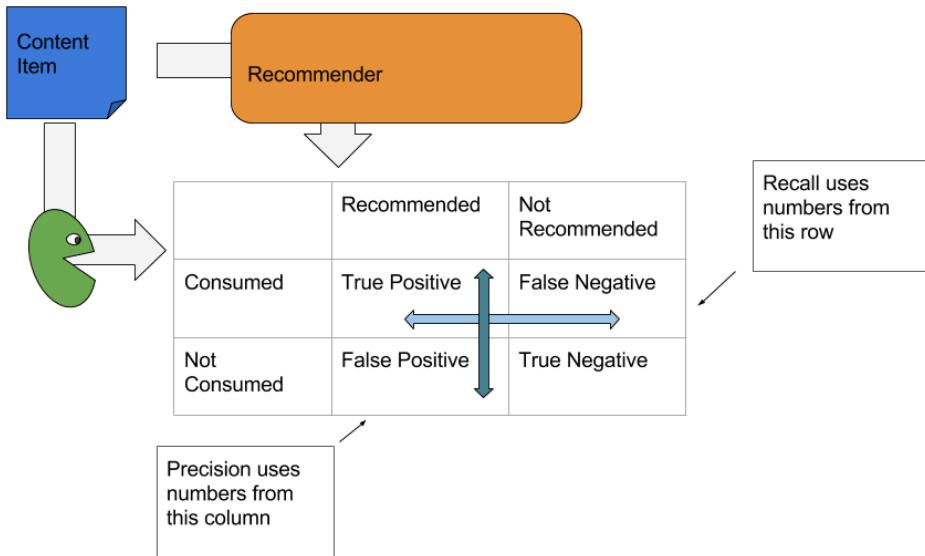


Figure 9.7 showing how precision and recall are calculated, comparing if users consumed the same things that has been recommended by the recommender.

Agreeing on the way to split the outcomes of your test. We can define two different metrics. Precision, which says that out of the recommended items what fraction did the user actually consume.

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

To balance this, we also look at recall. Which instead says that out of all the items that the user consumed was recommended.

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

Often a recommender system is implemented such at it should give the user at least one choice for what to buy/view next. It is therefore very important that there is always at least one relevant object in a top-n recommendation. It is there often considered more important to optimize precision, while not putting too much importance into whether or not the user gets all the possible relevant items (this measure was the recall)

You might ask, how to we translate into a metrics that we can use for all our recs. Decision-support comes from information filtering, originally used for calculating quality of search results. A search result can be very long, so in the world of recommenders they have restricted the measurement to looking only at the k top elements.

Precision at K

It is measured simply by saying number of relevant items between the first k items. "Relevant" is something that needs to be predefined. It could be hard boundary, saying that an item is relevant if the user has rated it above four stars, or you could say it must be more than one star from the users mean. If we are talking about implicit ratings, then it could also simply be items consumed, or something altogether different. But relevance needs to be decided on.

So, if we are interested in the precision within the first k items, we call it *precision at k* and defined as the following:

$$P@k(u) = \frac{\#\{relevant content in the top k positions\}}{k}$$

If you want to use precision, you can calculate the average over all users, by summing up all precisions and then divide by the number of users.

But if we have a top 10 then we would like to have relevant items, all over the place but most importantly in the top. This is why more and more are using Ranking metrics to evaluate their recommenders. Actually the first we will look at uses Precision at k.

MEASURING RANKING METRICS

The first item recommended is always the most important one, then the second and so on. So, when evaluating we should also take that into account. In the following we will look at different metrics which does exactly that.

Mean Average Precision (MAP)

The average precision (AP) could be used to measure how good the rank is. Simply by running the precision from 1 to m where m is the length of your rec (usually denoted k). So, to take it a step further we take the average of the precisions over first, the first element, then the two first... until we arrive at m . Like shown in figure 9.8. and written in the following formula:

$$AP(u) = \frac{\sum_{k=1}^m P@k(u)}{m}$$

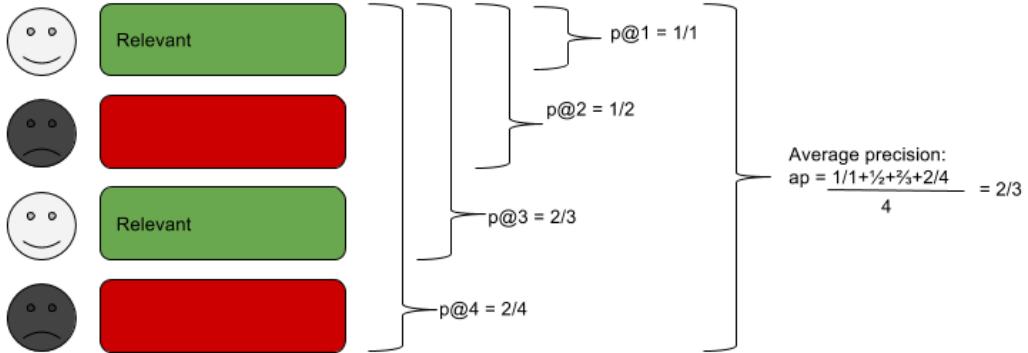


Figure 9.8 calculating the AP by taking the average of each precision@k for k being from 1 to length of the recommendation.

This works per recommendation, so if we want to use it as a measure to evaluate a recommender, we will then take the mean of the average over all recs.

$$MAP = \frac{\sum_{u \in U} AP(u)}{|U|}$$

Discounted Cumulative Gain

The discounted cumulative gain is hard to say, but easy enough to understand, basically it is about finding each relevant item and then penalize it the further down it is in the list. So, it is in a sense much like the MAP as we looked at above. Only where the MAP only works with relevant or not, the DCG looks at levels of relevancy. We give each item a relevancy score. In case of a recommender system it can be the predicted rating, or profit on the item. Relevance is also denoted as gain. So, we could also have called it the Discounted Cumulative Relevance. The relevance is discounted by the position in the list, these are added up so that's the cumulative.

$$DCG = \sum \frac{2^{rel[i]} - 1}{\log_2([i] + 2)}$$

DCG has an even more strict big brother called the Normalized Discounted Cumulative Gain

Normalized Discounted Cumulative Gain

The thing about DCG is that it is not easy to compare with DCG calculated in other evaluations, to get around that one can use the Normalized Discounted Cumulative Gain (NDCG) which, compared to the DCG is that it is put it as a proportion of the optimal ranking. So, we will have 1 if the ordering is optimal.

$$nDCG = \frac{DCG}{IDCG}$$

The nDCG is often used in competitions at Kaggle.com⁶¹

If you are interested in implementations of these in Python and many other languages its worth checking out the following GitHub repo: <https://github.com/benhamner/Metrics>

9.8.1 Performing the experiment

Well, okay, it's not that simple. First, we need to go into some tedious details about the data and how to divide it.

To start we need to figure out what data we have and whether or not it illustrates the reality we want to evaluate. For example, in many places in this book I have mentioned that you need some data to personalize, so if we have a lot of users with only few rating, then it's a bit evil toward the recommender to penalize it if it can't recommend something sensible to a user that we basically don't know anything about. Another problem that we haven't talked that much about is if you have too much data. In that case, you might need to do sampling.

SAMPLING

Sampling is about extracting a subset of the data that represents the same distribution communities and oddities as the full dataset. That can be quite hard. At its simplest form, it's a matter of randomly picking items for your subset, the more considered way of sampling is something called *stratified sampling*. If you have a data set with 10% men and 90 % women, then stratified sampling would ensure that your sample will contain the same distribution.

GOOD CANDIDATE USERS FOR THE EVALUATION

We should remove all users with few ratings. In chapter 7, we discussed that similarity would only really count if there was an overlap of more than 20 items. To allow for that, we need to cut off all users with fewer than 20. Back then, we also pointed out that 20 items was a lot. If you look at Figure 9.9 then you can see that a restriction of minimum 20 ratings will leave less than 10000 users.

⁶¹ <https://www.kaggle.com/wiki/NormalizedDiscountedCumulativeGain>

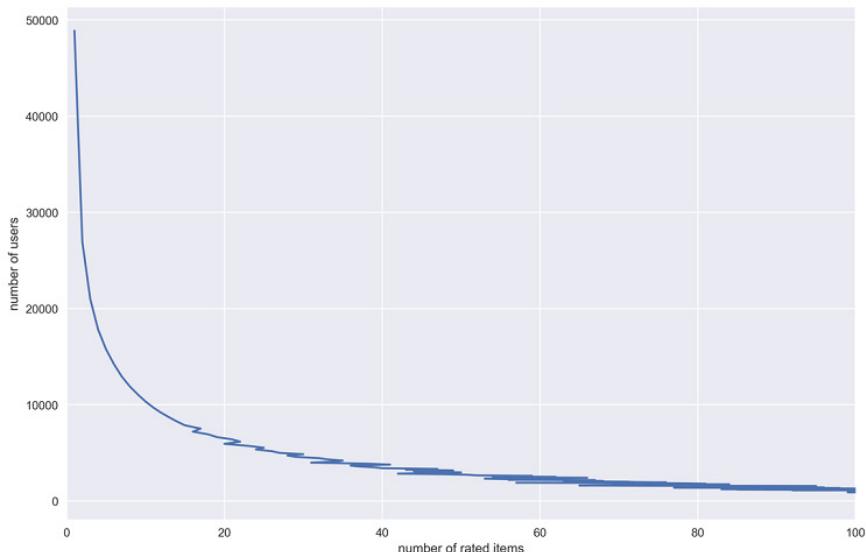


Figure 9.9 The plot shows how many users will remain in the dataset if putting a minimum number of ratings a user should have before its included.

As everything else, it is up to you and the rules of the domain to put the boundary. The algorithm won't work for users with only one rating. A user with only one registered ratings also won't do anything for collaborative filtering algorithms, which work on binding items based of users having rated them. On the other hand, removing all users who has rated less than 20 items also seems to try to paint a miss guided picture of the world. At least if you are making a recommender for something that has similar buying habits as book store or something like that. I am thinking that in most e-shops it will take some time before arriving at 20 items.

No matter what you decide then we need to split the data.

SPLITTING THE DATA INTO TEST, TRAINING AND VALIDATION SET

When performing the actual experiment, we need data for the recommender to train and calculate the predictions but we also need data to test if the predictions work. So, we will split the data. Actually, we need three datasets. So, we have the test set which is the set where we calculate how good the algorithm is at predicting and producing ratings. But if we keep correcting the recommender system to run better on the test set, then we will end up with a recommender system that is very good at predicting the things in the test set, but maybe not good at predicting new virgin material. So, the test set can only be used one time when you are finished optimizing the system. But that leaves us with the training set, what we should do is the split the training set further, such that it is a training set which you use to

create the model, and a validation/dev set, which you use to optimize the variables of the recommender.

In short, we should split the dataset into a test set and a training set. Then optimize the recommender (it could be the number of recs that should be returned, what should be the minimum similarity that should be saved etc.) by further cutting the training set into two (train and dev/validation set).

Optimize one parameter at the time.

When you have a optimized model, use the test-set to calculate the actual evaluation measure.

So let's split. But how is that done?

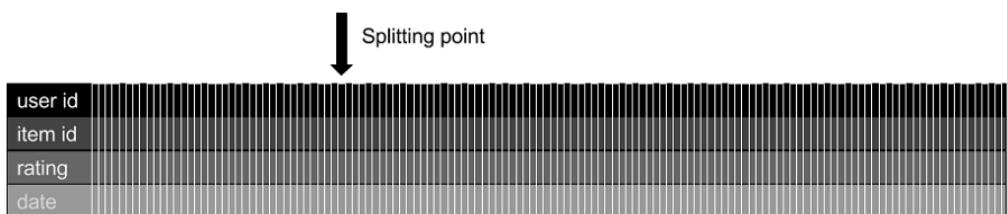


Figure 9.1023 The historical data will contain a long list of tuples each describing a rating done by a user on an item. How to split the data has a big influence on how the evaluation will go.

In the following we will assume at the test set is already cut out. To use any of the metrics we just spent a lot of time learning, we need two things – or more precisely, we need two data sets. We will get these by splitting our historical dataset into two parts. We will use one to teach the recommender algorithm and one for us to evaluate what it has learned. Before you find a saw and cut the dataset in two, though, let's consider what we are trying to do because that will affect how we should slice and dice the data.

Let's say we have a small dataset, I cut this out of the rating table in the MovieGEEKs site.

user_id	movie_id	rating
1	0068646	10.00
1	0113277	10.00
2	0422720	8.00
2	0454876	8.00
2	0790636	7.00
2	0816711	8.00
3	1300854	7.00
3	2170439	7.00

3	2203939	6.00
4	1300854	7.00

We will use it to show how the different splitting techniques works.

RANDOM SPLITTING

Often when you deal with predictive machine learning algorithms, it's normal to just pick a percentage p of the data and use that for training, then use the rest for testing the trained algorithm. There are many libraries that will take care of this for you (scikit-learn to name one⁶²), In a recommender system you have the problem that splitting ratings randomly will make the recommender train with ratings that are added after the ones that it needs to predict. Often recommenders don't distinguish between ratings done now or a year ago. But it might have an effect on its ability to predict ratings.

It's a good idea to split the data around 80-20, the 80-20 is not a law, it is important to have as much data to train the algorithm, but it won't matter if you done have a nice selection of data to verify the result. We would select two rows by random and take them out. Let's look at two examples where this wouldn't work well. Imagine the random selection was a bit lazy and just took two from the top:

user_id	movie_id	rating
1	0068646	10.00
1	0113277	10.00

The result would be that the recommender would not know anything about user 1 and therefore not be able to recommend anything. The same would be the case if you pulled out these two:

user_id	movie_id	rating
3	1300854	7.00
4	1300854	7.00

Now the recommender would not be able to find any similarity for movie 1300854 and it would therefore be lost. Taking out all the ratings of a certain value is not problem however.

⁶² http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

SPLIT BY TIME

From a recommender system evaluation point of view, it seems to make more sense to split the data based on a point in time, saying that everything before some point is used for training the algorithm. Only we can't with this data, because we don't have a timestamp, hence we would have to skip this option (or update the data model because the MovieTweetings dataset also contains timestamps).

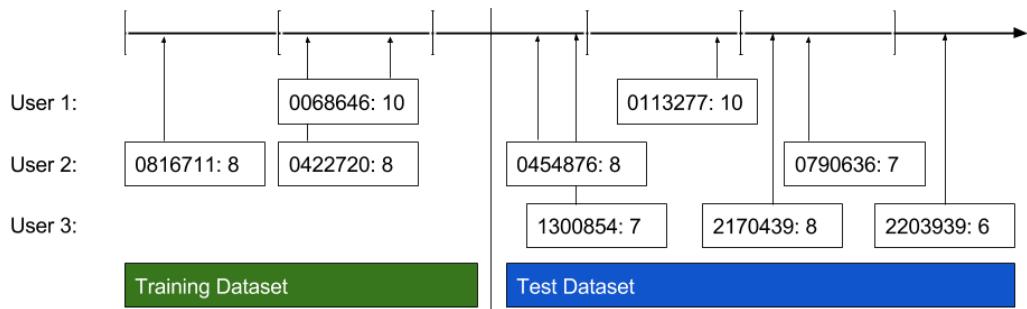


Figure 9.11 splitting data by time will create time snapshots of the data.

If we had the ratings above on a timeline instead as the brief example in figure 9.7 shows, we have a snapshot of how the data looks at a certain point in time. You can even test the recommender with a sliding point in time, and see how the recommender would have worked at any point in time.

There is a cost of splitting data like this; we will have users that only appear in the test set, which the recommender has no idea what to do with. Some would say that it is an excellent chance to see how it works with cold users, but for the evaluation of one algorithm it doesn't seem like the right thing to do. If, on the other hand, you are testing a hybrid that is implemented to handle cold-start users, then this is perfect. Alternatively, you can clean the test dataset for users that do not appear in the training set. An extreme of this method is that you iterate through all ratings and attempt to predict it, using only using ratings with timestamps that is earlier in the current rating.

SPLIT BY USERS

The last option we will look at doesn't divide the users between test and training sets. Instead, we will divide each user's ratings between a training set and a test set. The ratings will be divided by taking the first n ratings into the training set and the rest in the testing set. This is illustrated in the following table.

	user 1	user 2	user 3	user 4
1	0068646	0422720	1300854	1300854
2	0113277	0454876	2170439	
3		0790636	2203939	
4		0816711		

Here we put the two first ratings of each user in the training set; this means that user 1 and 4 don't have any ratings in the test set, but this way all users will be at least in the training set, and you won't find any users in the test set that haven't been seen before.

It is a bit more demanding to split data this way because you need to order each user's rating. If you have timestamps you can do something like the following:

Listing 9.4: SQL get the first two ratings of all users

```
select *,  
       ( select count(*)  
           from rating as rating2  
          where rating2.timestamp < rating1.timestamp  
            ) as rank  
  from Rating as rating1  
 where rating1.user_id = 1  
   and rank < 3
```

- ① For each row in the resulting table. Query the data to fount how many ratings were done before this timestamp.
- ② count all the ones with timestamp lower than the timestamp of the current rating.
- ③ filter by user_id, to split the data this needs to be done for each user.
- ④ only get the 2 first ratings per user, as shown in the table above.

If you have a lot of ratings, this will take a lot of time and you might be better off storing the number when you save the rating.

Dividing the data like this, is called the "*given n*" protocol; it is mentioned here because it is often used in research papers. Others would argue that this is the best way, even just to leave on rating for test for all the users.

CROSS VALIDATION

No matter how you split the data there is a risk that you will create a split that is favorable for one recommender. To mitigate this, try out different samples of the data for training and find the average and the variance of each of the algorithms to understand one is better than the other.

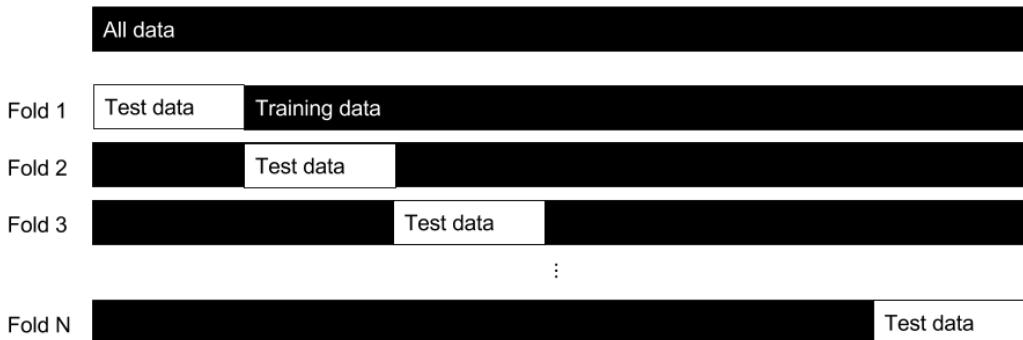


Figure 9.12 When you do K - fold cross validation you divide your data up in chunks

K-fold cross validations works by dividing the data into k folds then use k-1 folds to train the algorithm. The last fold is test data; you walk through the full dataset allowing each fold to be the test set. You run the evaluation k times, and then calculate the average of them all.

But the question from above still remains – How to we split the data? Do we just divide the ratings into k different folds? But then we have the same problem, that we might end up with users that are only present in one set. Instead we will take all the user IDs and split those into k different piles. Then, though, we will have the problem of users on being in only one fold. To solve that, divide the ratings of the test users such that for example, 10 ratings go into the training part and the rest are used for testing. Let's try to implement this.

9.9 Implementing the experiment

Let's try to implement an evaluation. As always we will look at an implementation in the MovieGEEKs website. Which can be found at

<https://github.com/practical-recommender-systems/moviegeek>

If you haven't done it yet, I recommend that you download it and try to run it on your machine. The dataset we use is one called MovieTweetings which can also be found on github⁶³, but if you are downloading the website there is a script you can run to get all the data you need. For instructions to get the MovieGEEKs site running please refer to the readme file.

⁶³ <https://github.com/sidooms/MovieTweetings>

9.9.1 What we will implement

We will do a k-fold cross validation where $k = 6$. The first 10 items of the test users go to training the algorithm. In this chapter, we will describe an evaluation runner framework and show how we can evaluate the algorithm from chapter 8. And then use the same throughout the rest of the book for each of the algorithms we describe. We will evaluate it using the Mean Average Precision. Figure 9.12 shows the pipeline we will walk through. It will go through the following steps:

- Clean the data
- Split the users into k-folds
- Repeat for each of the five folds
 - Split data
 - Train recommender.
 - Evaluate recommender on test set.
- Aggregate result

We will look at each in the following. The code for this can be found on github in the folder that is called builder. The file is called `evaluation_runner.py`.

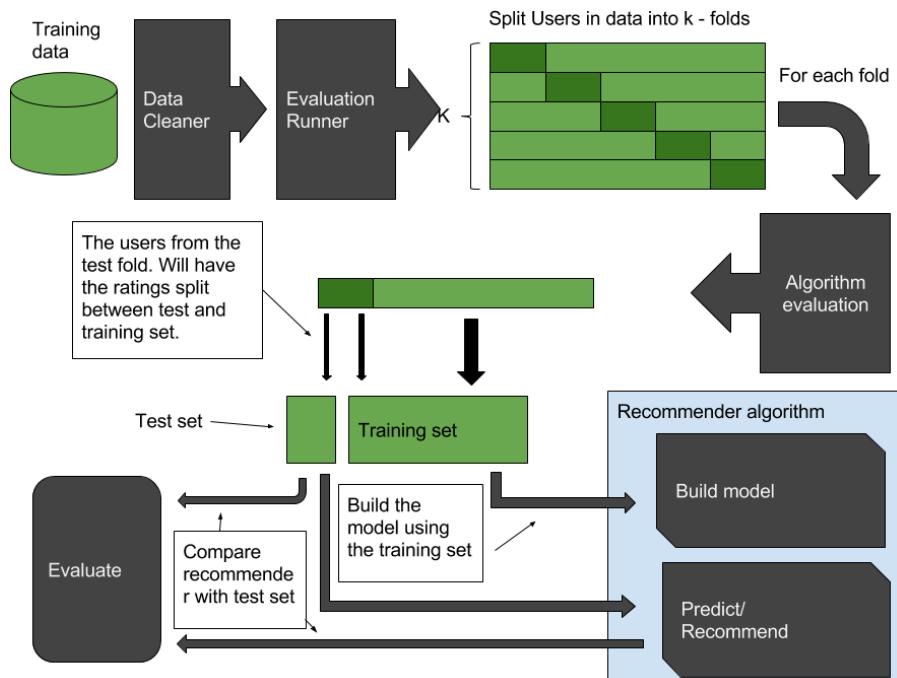


Figure 9.13 The evaluation pipeline.

CLEAN DATA

Most datasets need a bit of housekeeping before you evaluate them. You can say that maybe it is cheating, but for your own shake its better to do it. As big datasets take forever to test, and in our case then all the users who only rated a few items, doesn't help the recommender much and the chance of recommending them something sensible is very small. In my code I have chosen to do as shown in the following code:

Listing 9.5: Builder\evaluator.py

```
def clean_data(self, ratings, min_ratings=5):

    user_count = ratings[['user_id', 'movie_id']]
    user_count = user_count.groupby('user_id').count() ①
    user_count = user_count.reset_index()
    user_ids = user_count[user_count['movie_id'] > min_ratings]['user_id'] ②

    ratings = ratings[ratings['user_id'].isin(user_ids)] ③

    return ratings ④
```

- ① We need to count how many items each user has, so we can cut away the ones with less than the minimum number of ratings a user should have.
- ② When you do a groupby (as we did above) the you need to reset the index to access the column by name.
- ③ filter the list of user ids to remove the users with less movie ratings than min_ratings
- ④ Now filter the ratings to only contain ratings from the users we filtered above.

Now that we have the data cleaned and shiny we are ready to proceed.

SPLITTING THE USERS

Splitting the users into the k folds is not really that hard, so we wont go into much detail here, only show we use scikit-learn tool.

Listing 5: evaluator\evaluation_runner.py

```
def split_users(self, users, num_folds = 5):
    kf = KFold(n_split = num_folds) ①
    return kf ②
```

- ① Create an instance of the k folder from sklearn⁶⁴, and initialize it to do num_folds folds
- ② kf has a split method, which when called will return a training set and a test set, corresponding to the next configuration of the k folds.

To get any benefit of the splitter we will use it in a for loop, which runs the steps shown in Figure 9.11 in the box called repeat n times.

⁶⁴ http://scikit-learn.org/stable/modules/cross_validation.html

Listing 6: evaluator\evaluation_runner.py

```

def calculate_using_ratings(self, all_ratings,
                           min_number_of_ratings=5,
                           min_rank=5):

    ratings = self.clean_data(all_ratings, min_number_of_ratings)

    users = ratings.user_id.unique()
    kf = self.split_users()

    validation_no = 0
    paks, raks, maes = Decimal(0.0), Decimal(0.0), Decimal(0.0)

    for train, test in kf.split(users):
        validation_no += 1
        test_data, train_data = self.split_data(min_rank,
                                                ratings,
                                                users[test],
                                                users[train])           ①

        if self.builder:
            self.builder.build(train_data)                      ②

        pak, rak = PrecisionAtK(self.K,
                               self.recommender).calculate(train_data,
                               test_data)          ③

        paks += pak
        raks += rak
        maes += MeanAverageError(self.recommender).calculate(train_data,
                                                               test_data)      ④

        results = {'pak': paks / self.folds,
                   'rak': raks / self.folds,
                   'mae': maes / self.folds}                         ⑤

    return results

```

- ① The kf.split above gives us the user_ids but we still need to split the data of the test users.
- ② the builder, is a class reference to a recommender algorithm trainer. Calling build means that it will prepare for doing recommendations. In the neighborhood algorithm it means that it will build the item similarity matrix.
- ③ The Precision at K evaluation. Which will be run on the top 5 recommendations. Using the neighborhood recommender.
- ④ Mean average Error is calculate
- ⑤ return the result, averaging over the number of folds.

To e Before looking at the Precision at K and the mean average error we will quickly look at how to split the data.

SPLIT DATA

As we talked about above, we have k folds, where all ratings from the k-1 folds goes directly to the training set, the last fold is divided such that all items which are ranked lower than min_rank is in the test set and the rest is put in the training set.

Listing 7: evaluator\evaluation_runner.py

```

def split_data(self, min_rank, ratings, test_users, train_users):
    train = ratings[ratings['user_id'].isin(train_users)]  
①
    test_temp = ratings[ratings['user_id'].isin(test_users)]  
②

    test_temp['rank'] =  
        test_temp.groupby('user_id')['rating_timestamp'].rank(ascending=False)  
③
    test = test_temp[test_temp['rank'] > min_rank]  
④

    additional_training_data = test_temp[test_temp['rank'] >= min_rank]  
⑤
    train = train.append(additional_training_data)  
⑥

    return test, train  
⑦

```

- ① creating a dataframe with all the ratings of the train users, which are the ones from the k-1 folds.
- ② creating a dataframe with all the ratings of the test users, which are the ones from the last fold
- ③ for each user, rank the content items ordered by timestamp, so the item with rank 1 is the newest and so on.
- ④ remove all items which rank higher than min rank
- ⑤ take all the items from the test users with ranks higher or equal to min rank
- ⑥ add them to the train data.
- ⑦ return the two dataframes.

9.10 Evaluating the test set

So, having done all these things, I am afraid the sad truth is that we wouldn't have won the Netflix prize with the algorithm implemented in Chapter 8. But remember that was also what they said at Netflix after they cashed out one million dollars. They said that it was not a good way to evaluate if a recommender is good.

The next brutal thing is that there are a lot of parameters which you can adjust to make it look better. You can restrict the users that are in the test. We can also look at the do precision at 10 or at 100. So, rest assured that everybody has received some disturbingly low numbers the first time they ran an evaluation on their recommender (I got a precision that was something like 0.063 the first time around).

I guess I sound a bit like a sour loser, and yes, I was disappointed. But what you are looking for is a benchmark number. This is what you should do, if it is too close to zero, that means that it was not able to predict basically any of the films that the users had in their test set. And that is bad. If you on the other hand have something that is almost one it means that all users in your test set receives recommendations which all contain items which the user has rated positively.

9.10.1 Starting out with the baseline predictor

But before actually evaluating your new recommender, you should start out evaluating it on a very simple recommender, for example just on one that always recommends the most popular items, and see what numbers comes out. Then you have something compare with.

Listing 9.8: recs\popularity_recommender.py

```

class PopularityBasedRecs(base_recommender):

    def predict_score(self, user_id, item_id):
        avg_rating = Rating.objects.filter(~Q(user_id=user_id) &
                                         Q(movie_id=item_id)).values('movie_id').aggregate(Avg('rating'))      ①
        return avg_rating['rating__avg']

    def recommend_items(self, user_id, num=6):
        pop_items =
            Rating.objects.filter(~Q(user_id=user_id)).values('movie_id').annotate(Count('user_id'))      ②
            , Avg('rating'))
        sorted_items = sorted(pop_items, key=lambda item: -
            float(item['user_id__count']))[:num]      ③
        return sorted_items

```

- ① the predict score method will just find all the ratings given to the particular item, and average that.
- ② the top n recommender method will return the items with which has been rated most.
- ③ and return a sorted list, based on how many users rated it.

Start by running this. If you are measuring according to the MAP we talked about above, then if you test the most popular rec with $K = 2$ and then jump 2 until 20, the result is shown in figure 9.14. In figure 9.15 there are the same chart but for the neighbourhood collaborative filtering algorithm we saw in chapter 8.

The first evaluation (of the popularity recommender) was done by executing the following script:

Listing 9.9: script to evaluate the popularity recommender

```
>python -m evaluator.evaluation_runner -pop
```

The next chart, shown in fig 9.15 was done using the data which was create by running almost similar code

Listing 9.10: script to evaluate the popularity recommender

```
>python -m evaluator.evaluation_runner -cf
```

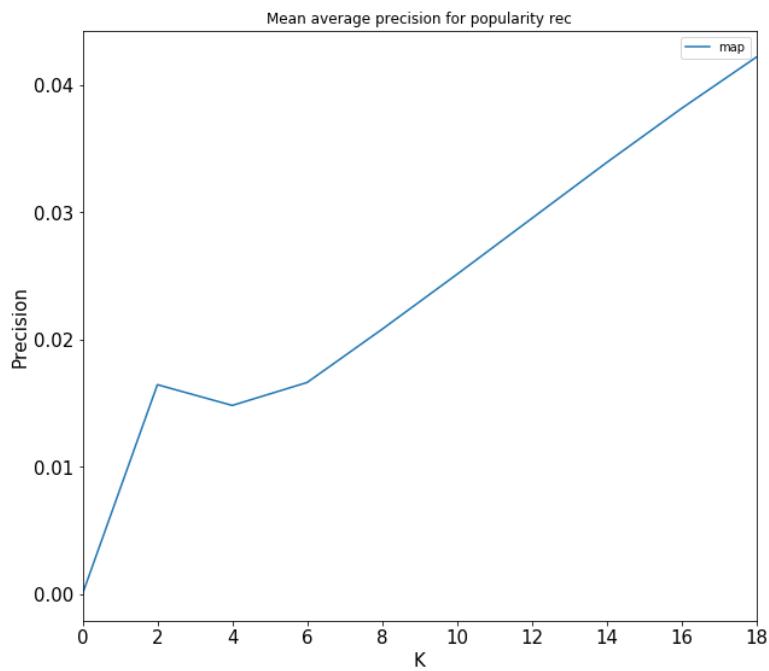


Figure 9.24 The mean average precision for the popularity recommender. There is an interesting hill at TOP-2 and then a slight decrease for TOP-4 recs.

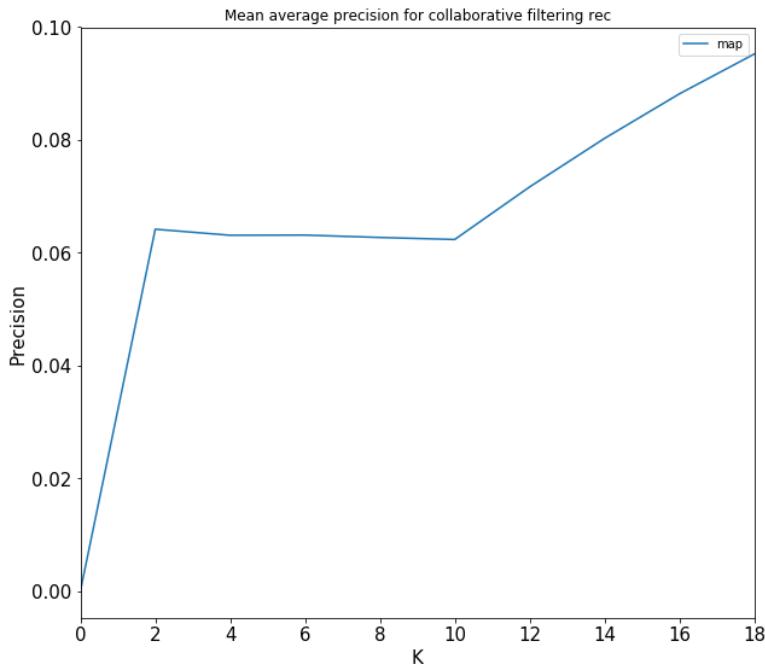


Figure 9.15 Mean average precision for the neighborhood model which we looked at in chapter 8. It is easy to see that we should choose this over the popularity model, as the MAP is much higher here. The problem here is that to get this precision we need to only look at users who has rated 20 items,

So even if the neighborhood model might not be good for recommending things to all users, it is still worth implementing it and use it out popularity, since it has a higher precision. Probably as an easy start you could implement a hybrid recommender which will return results from the neighborhood model, and when the user has too few ratings, it could spice it up with results from the pop recommender.

9.10.2 Finding the right parameters

So we should start out taking away the test set. Then we should look at the training set, which is then split further into smaller parts when we want to optimize the parameters of the model. The model we implemented in chapter 8 has the following parameters:

Listing 9.11: evaluator\evaluation_runner.py Model Parameters

```
min_number_of_ratings = 20
min_overlap = 5
min_sim = 0
K = 10
min_rank = 5
```

- 1
- 2
- 3
- 4
- 5

- 1 require that each user has rated a minimum of 20 movies
- 2 We will only look at similarities where more than min_overlap users has rated both movies.
- 3 We will only save similarities which are bigger than min_sim
- 4 We will give the 5 newest ratings to the test set.

We want these parameters to be set, such they return the best evaluation. So, to do that, pick one of them, and then run the evaluator for a range of values. For example, the min_number_of_ratings parameter says something about how many ratings a user should minimum have before been considered in the evaluation.

Now do a simple for loop like the following:**Listing 9.12: evaluator\evaluation_runner.py**

```
builder = ItemSimilarityMatrixBuilder(min_overlap, min_sim=min_sim) ①

for min_overlap in np.arange(0, 15, 1):
    recommender = NeighborhoodBasedRecs() ②
    er = EvaluationRunner(0, ③
        builder,
        recommender,
        K) ④

    result = er.calculate(min_number_of_ratings,
        min_rank,
        number_test_users=-1) ⑤

    user_coverage, movie_coverage = RecommenderCoverage(recommender).calculate_coverage() ⑥
```

- 1 create an instance of the model training class, in this case it is the item similarity matrix builder.
- 2 run through a range of values, where we are trying min_overlap is being tested
- 3 create a new instance of the recommender
- 4 create an instance of the evaluation runner
- 5 run the evaluation calculator
- 6 calculate coverage.

When that is finished and you believe that you have an optimal number there, you can move on to the next parameter to be used. Go through all of them, and if you really want to do a good job, you can run through it all a couple of times in different orders.

I am afraid that there are some manual steps in testing each of the parameters, you will have to fix the code such that the thing that is iterated over is the parameter you are training.

I am afraid that this is it for offline experiments. I hope you got an understanding about how you do, but also how many places it can go wrong. It is still a good idea to do it, so you can measure if you do improvements, but remember to evaluate on the right things.

9.11 Controlled experiments

Controlled experiments are done by inviting real humans to perform a test in controlled environment. You can invite people to follow a checklist; in the case of our movie site, the goal would be to have users insert their preferences (rate some movies), view if the

recommendations are good, show two different types of recommendations to the user, and see which seems to work better.

The good thing about this is that you can monitor user behavior. You can ask users questions about what they thought. The downside of this is that users might behave differently in a controlled environment than they would if they were alone. It might also be quite time consuming and difficult to set up an experiment that you can learn from.

9.11.1 Family and friends

A way to do something similar is to provide access to a small group of people, which is normally termed family and friends, who you trust to try out the system and return honest feedback. This alternative will make everybody an expert in recommender systems who will tell you how and why you need to update the recommender... But when it happens, remember to bow your head and listen to their feedback. Because even if they don't understand the new super complicated algorithm that you are testing, then they are still representatives of the end users.

9.12 A/B testing

A lot of things can influence how things are going on your application. You might have an increase in conversions because you just added some new content to your catalogue, or Christmas is around the corner so people who celebrate Christmas might go on a frenzied buying spree, or some completely different reason. What I am getting at is that there is probably a lot of different factors that influence the state of things, including a lot of things that you are not the master of. This is important to consider, because you might launch a new recommender system, or a change to an existing one, just when something else happens which might make it look like it's the recommenders fault. It is hard to test if a recommender has a positive effect or not. To get around this you can use A/B testing to see if your change has any effect.

You can test a new recommender algorithm by redirecting a small part of the traffic to the new recommender, and letting the customers unknowingly indicate which is better. In fact, usually it's business as usual for the customers. They won't know that they are part of an experiment. That's the whole point.

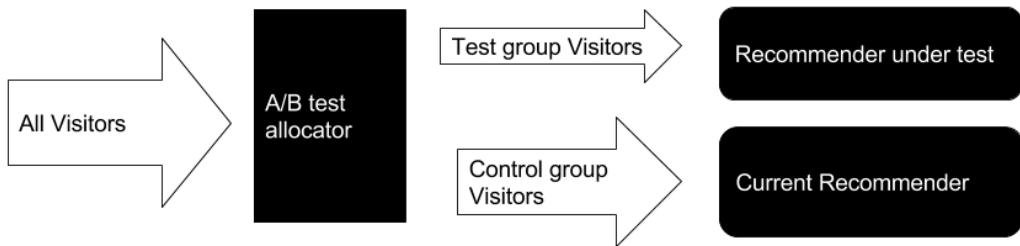


Figure 9.16 In an A/B test visitors are split into two groups, the test group who sees the new feature and a control group who will continue as usual.

How does it work? Let's say you just finished some new algorithm implementation, the offline evaluation looked okay, and it is now time to step it up so you implement an A/B test. The test will show if there is a significant difference between the current recommender and the new one. In practice, it works as shown in Figure 9.15 by diverting a small percentage of traffic to the new feature.

A concrete example could be that you want to test out two different settings of the neighbourhood size or you want to test if one recommender performance better than another. Registering all behaviour of the visitors in the test group (as well as in the control group) will enable you to decide whether it is worth it to upgrade all traffic to the new algorithm.

One of the risks to consider with A/B testing is that if your new feature is not good, then you might risk losing customers as they would experience a drop of quality. But there is always a price for a chance to do something better. On the other hand, if you put new features in production without testing them you might end up being in much more trouble. At companies with many millions of users, they A/B test everything before it goes into production⁶⁵

A/B testing is something you'd want to look more into as it will be the basis of much feature development for data driven applications in the future.

One thing is A/B testing, and you need to do that, but features that test well might not be good in the long run, it is therefore a good idea to keep running tests, one way of doing that is Exploit/Explore which we will briefly summarise in the following section

9.13 Continuous testing with exploit/explore

A/B testing is about deciding whether or not to deploy a new feature. It would be much easier instead to be able to say, "I have these two algorithms running, and I believe that sometimes

⁶⁵ read more about netflix's A/B test here: <https://medium.com/netflix-techblog/its-all-about-testing-the-netflix-experimentation-platform-4e1ca458c15>

one is better, but other times it's the other way around," and make the computer figure out which it should use? That is exactly the explore/exploit idea: that you can either exploit the knowledge you have gained so far and use the one that you think is the better or you can explore another feature which the system doesn't know so much about. You can consider this as continuous A/B testing. Another way of considering it is like you have a long row of one-armed bandits (aka slot machines) as shown in Figure 9.16. As a trained gambler, you might know that machine 1 and 2 often gives out something but not so much which the others you don't know much about. Should you then go for the safe bet and put your money in 1 or 2 or should you be daring and try something new. This problem is similar to the recommender system who knows that popular items are often good, but you might gain more by simply showing something new. This can be used both when choosing between algorithms, but also down to items. For more on this subject, I recommend looking at a book called *Statistical method in Recommender Systems*.

Exploit/Explore is also used at Yahoo news to introduce new content, such that new content doesn't stay cold products, but is tested out among users and thereby allows the system to understand what users response is to the new content.

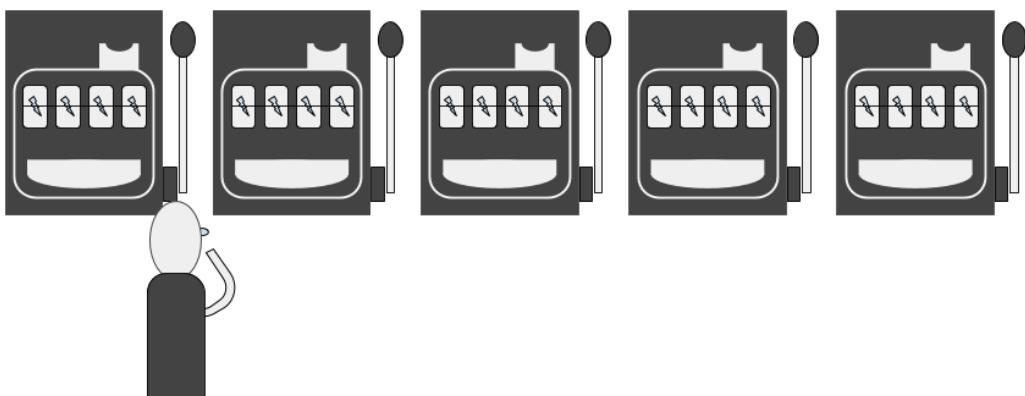


Figure 9.17 Exploit/Explore is often explained as the problem of a gambler who needs to choose the next slot machine to put a coin in

9.13.1 Feedback loops

Collaborative filtering uses user's behavior to create recommendations, but when you put your system in production, then you should also ask what the recommender does to the user's behavior, if the recommender works it might hinder diversity as we discussed back in 9.4.2. At NIPS (Conference on Neural Information Processing Systems) 2016 an interesting article was presented, which talks about how to measure this, called *Deconvolving Feedback Loops in Recommender Systems* by Sinha et al. it's a good idea to keep feedback loops in mind. Basically, it's about ensuring that users are given alternative ways to get data into this loop. If

you look at figure 9.17, the loop represents what happens if users only consume things that is shown as in recommendations. It is something that Netflix has to consider, since everything is a recommendation on their platform⁶⁶. Somehow you need to introduce new items into this loop. In figure 9.17 the X could be search, or manually added content, or adding random items in the feed, using armed bandits as we talked about above.

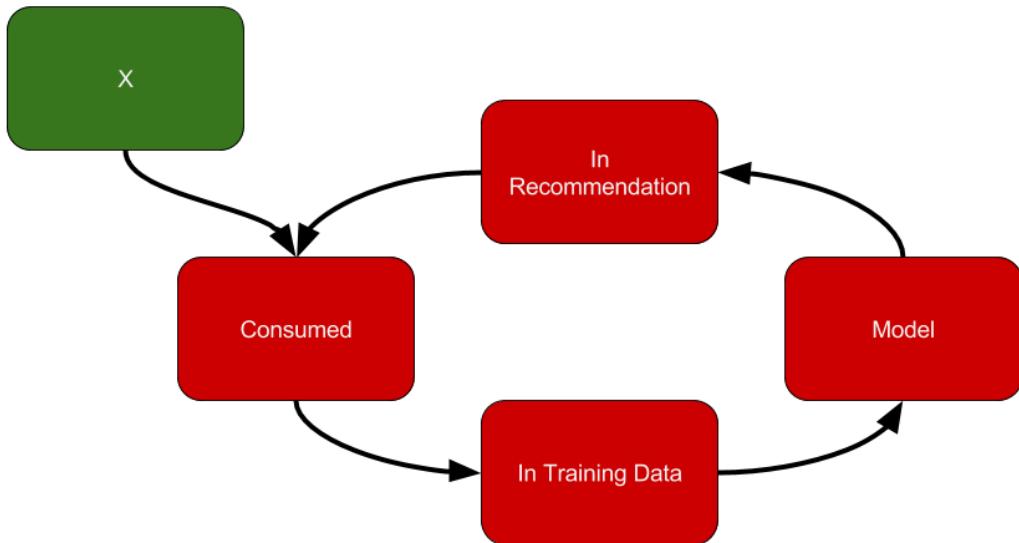


Figure 9.18 The feedback loop of recommenders

Just remember if you want diversity, you need also to enable the users to be divers.

9.14 Summary

So isn't this a surprise we have spent almost a whole book on talking about how to implement recommender systems, and now it turns out that it is actually just as hard to evaluate if they work or not. Evaluating recommender systems is hard. And it is a lot about choices, because you can either choose to measure prediction power or rank power as we talked about, or you can split your evaluation data into two sets in so many ways, either by random split, time split or user splits.

⁶⁶ Have a look at following slides. <https://www.slideshare.net/justinbasilico/deja-vu-the-importance-of-time-and-causality-in-recommender-systems>

What you should remember is that it is hard to do one run and then understand if it is good or not, it is more important to do the evaluation and then use it as base for future evaluations.

In this chapter, we talked about

- You should consider testing before you implementing the recommender system
- Regression tests guard your code against mistakes added unknowingly
- Serendipity is hard to measure, but is important.
- Different metrics are used to calculate if the recommender is good or at least if it compares to some baseline.
- A/B testing is something you really need to consider if you want to tune your recommender system. An A/B test can also be done to test which parameters works better, for example the size of the neighbourhood in collaborative filtering or the number of latent factors in matrix factorization.
- Explore/Exploit method is a really important subject if you want to keep optimizing your system when it is online.
- Watch out for feedback loops.

10

Content-based Filtering

In this chapter the content will be:

- You will be introduced to content-based filtering
- You'll learn how to construct user and content profiles
- To create content profiles, you will learn to extract information from descriptions using term frequency-inverse document frequency and Latent Dirichlet Allocation (LDA)
- You'll implement content-based filtering using descriptions of films in movieGEEKS

10.1 Introduction

In the previous chapters, we have seen that it is possible to create recommendations by focusing only on interactions between users and content (e.g., shopping basket analysis, collaborative filtering). It looks like it works nicely, but what about the things we know about the content? For a movie that can include things like genres, starring actors, directors, clothes sizes for clothes, and colors and engine size for cars. Can you really call a recommender system good if it doesn't take those things into account?

The answer is "YES!" as we have seen in the previous chapters, but it still seems like we are missing something or losing out on some information. In this chapter, we will try to make up for that, as this will be all about what we know about content and users' tastes.

By the end of this chapter, you should have a clear idea of how to build a content-based recommender and see how you actually build one. We will look at feature selection and how to process text to be used for the content filtering. We'll also look at two different algorithms called Term Frequency –Inverse Document Frequency (TF-IDF) and the new hot baby on the block: Latent Dirichlet Allocation (LDA).

Sounds exciting, doesn't it! Let's start out with an example to set the stage.

10.2 Descriptive example

On an average day, a conversation about movies could go something like the following:

Me: I just saw Ex-Machina (ok still haven't watched it but I look forward to it)

Imaginary interest person: Really, was it good.

Me: Yeah, there were some very interested subjects (Imagining that I watched it)

Imaginary interest person: Alright, so you like robot people.

Me: well yes (Feeling like I shouldn't say yes)

Imaginary interest person: Technology that goes bad, then you must like Terminator.

Me: Yes (relieved)

So, how did this happen? The *Imaginary interested person* thought about categories containing Ex-Machina, found the category "Robots that goes insane" and mentally looked up other movies in that genre (or maybe that is not an accepted genre so let's call it a category) and found Terminator. In this chapter, we want to implement a recommender which does the same.

Content-based filtering can be used to create *similar items* recommendations, which are also sometimes called "*more like this*" recommendations (see Figure 10.1), or to provide personal recommendations based on taste.

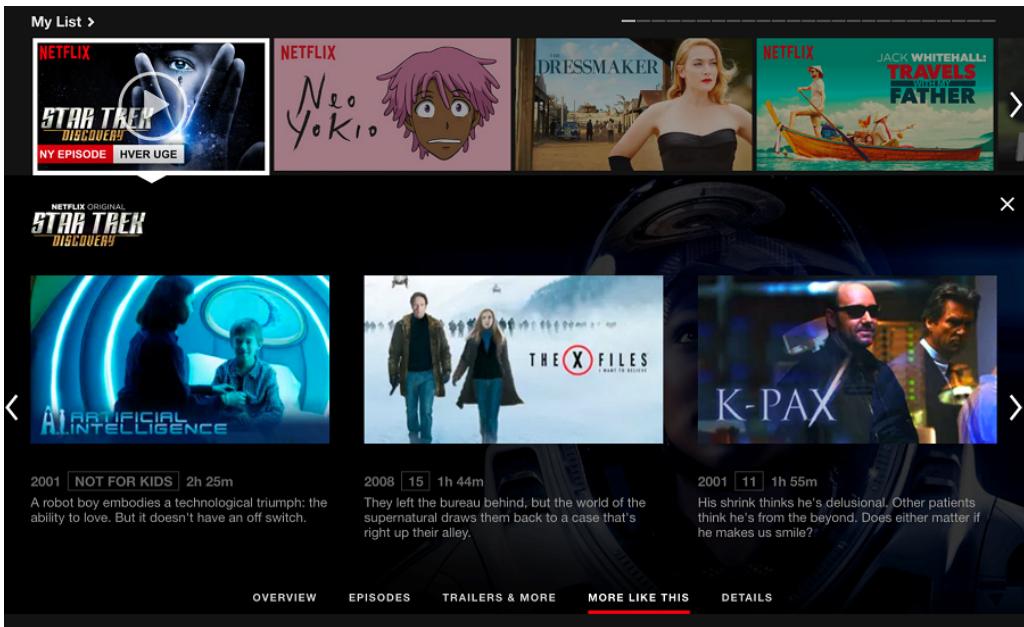


Figure 10.1 More like this recommendation from Netflix

The *Imaginary interested person* never used any opinions about whether the movies were good or not, she just used the metadata of the content being discussed and recommended based on that. To draw it, it would go something like figure 10.2.

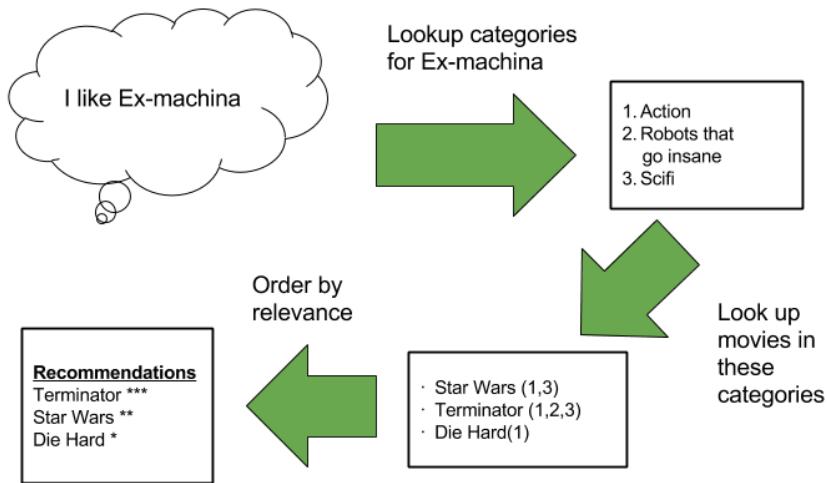


Figure 10.2 Example of content-based recommendation pipeline

That seems pretty clear (at least to the imaginary interested person and me), but if we would have to implement this into a recommender system, how would we get it to do the same?

In the example above (shown in Figure 10.2) we took one film and found a recommendation based on that. To do that, we will need to look for some way to find content that humans think is similar, and that can be a bit hard, as we are quite strange machines. But let's make an attempt. We will start out talking a bit more about what is content filtering when we will look at a way to find important words using the TF-IDF algorithm. Using that and the other tricks to extract features we will try to build a model where similar documents are close.

In this chapter, we have a bit of road to follow, so let's see if we can create some directions to start with so you know where you are going.

First, we will start out with a quick overview of what content-based filtering is.

Then we will look at some of the many ways to describe content. One way that is often used on the net is tags. We will start out looking at these tags. Tags are something that came out of the social internet or web 2.0 (even if it had been there for quite a while) and basically mean that users of a website add keywords to your content. An example of such is seen on IMDB as shown in figure 10.3. Another way to describe content is using textual descriptions, which is the next topic we will take up.

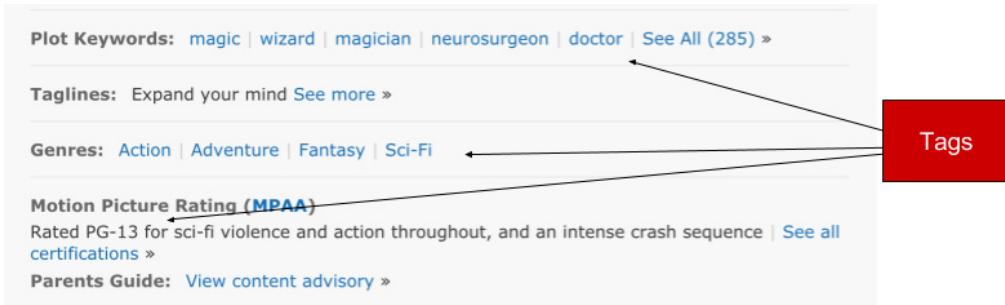


Figure 10.3 Screen cut from IMDB.com showing what we consider tags.

It is very hard to make computers understand full text, so we will look at how to extract things that are important while removing words which would just produce noise. There are many ways of removing noise words; we will look at some then we will calculate important words using the Term Frequency – Inverse Document Frequency (TF-IDF) method. We can use the important words found to create a list of categories (also called topics), which capture similar trends in the descriptions, by using an algorithm called Latent Dirichlet Allocation. The term “latent” refers to the fact that the topics found do not really compare to any category that we usually know, the Dirichlet is the way the documents are described using these topics and finally Allocation, means that words are allocated into topics. This is not the complete truth, but without going into a lot of statistics, then it is a good way of remembering it.

When we have all these things clear, we will of course look at how it is implemented in the MovieGEEKs app.

Now to content-based filtering.

10.3 Content-based filtering

Content-based filtering seems a bit more complicated than collaborative filtering because it's about extracting knowledge from the content. We will try to extract precise definitions of each content item and represent each item as a list of values. It sounds pretty easy but it does pose some challenges. Figure 10.3 illustrates a simple version of how to train a content-based recommender.

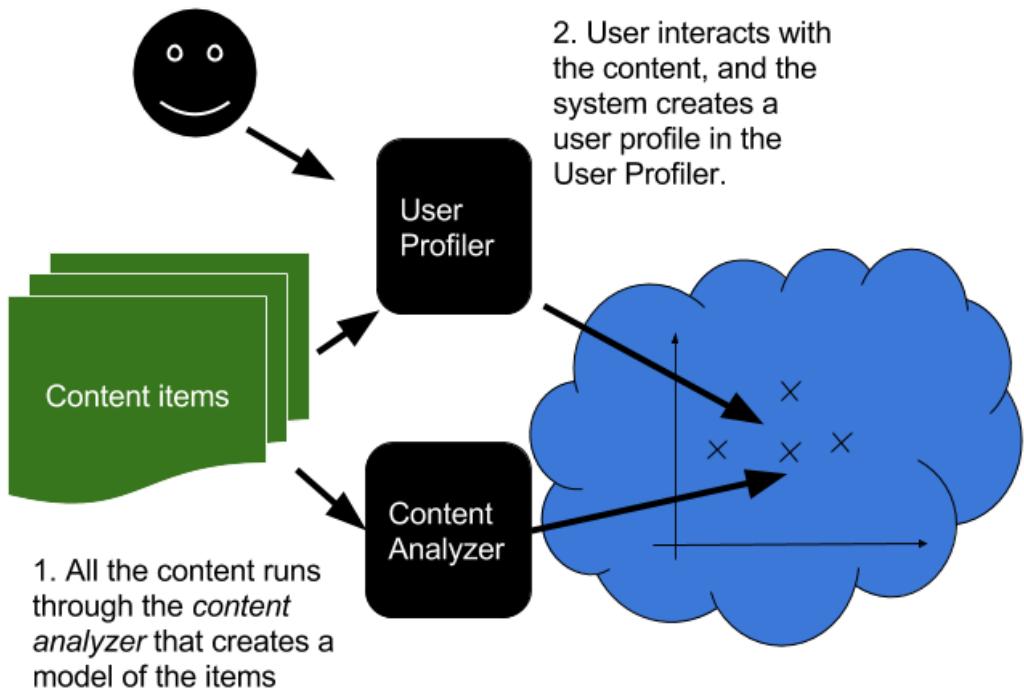


Figure 10.4 training a content base recommender

While Figure 10.4 shows how it is used when a user arrives. So to sum up, what I loosely said before, we need the following to make it work:

- **Content Analyzer:** creates a model based on the content items. In a way it will create a profile for each item. It is where the training of the model is done.
- **User Profiler:** creates a user profile, sometimes the user profile is just a list of items consumed by the user.
- **Item retriever:** retrieves relevant items found by comparing the user profile to the item profiles as shown in figure 10.5. If the user profile is just a list of items, then this list is iterated and similar items are found for each item in the users list.

There are several ways to implement these steps, and this chapter is about how it will work. We will look at each of the 3 points in turn.

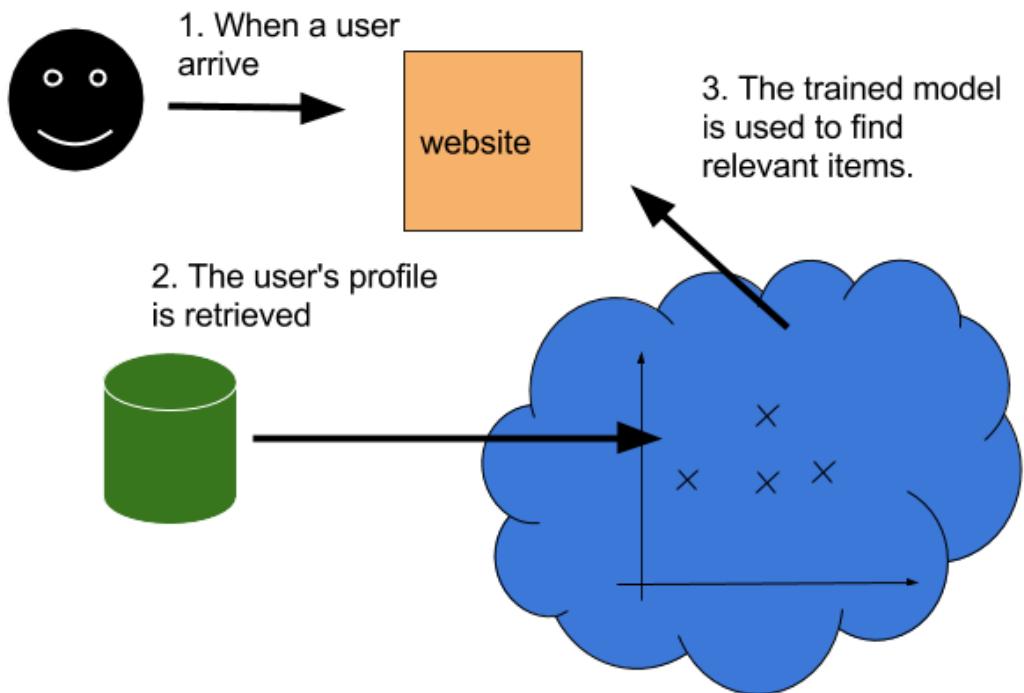


Figure 10.5 The item retriever returning recommendations with content based filtering

10.4 Content Analyzer

The Content Analyzer is the poor soul that will be given descriptive data about our content and have to map it into something the machine can use, like a vector.

To implement a Content Analyzer, we need to chat a bit about content: What is it and how do we understand it? In other words, we need to talk about feature extraction. Which is the topic of extracting things that we think is important for our algorithm to work.

10.4.1 Feature extraction for the item profile

Data about data is called metadata. To avoid confusion we call data about films metadata as well. So metadata about a film is everything that you can find on an IMDB page, such as genre, starring artists, and production year. It could also be something like the style of filming or types of clothes worn by the actors in the film, or in other domains, the shade of paint on the car or amount of freckles on men on dating sites.

I like to split the metadata loosely into two types:

- Facts
- Tags

This is not a division normally used, but it will be beneficial for you to think about the two. Because the facts are the things that can't be disputed like production year or starring actors in a movie, that means that you can also use them as input. On the other hand, the tags can mean very different things to people and should therefore be considered before adding them. The social internet has made it very popular for people to add descriptive tags to content. Tags can be something simple like "uplifting" or something more subjective like "Breaking The Fourth Wall"⁶⁷. "Breaking The Fourth Wall" is one used to describe *Deadpool*. No idea what it means, but 10 people said that it was relevant, and apparently, it applies to a number of films across different genres and decades⁶⁸.

Another challenge with the tags people put on films, is that people have very different ways to express themselves. A very simple example is how people talk about a James Bond film. I would probably say it's a "Bond film" and tag it as such if I were a tagger. But looking at the data, you can see that there are several different ways, predominantly using "007". To make our system understand that people are in fact talking about the same segment of films, it would be worth streamlining the tags to use the same word for the same thing as much as possible. Ideally, we would also like to split tags that mean different things to people.

There are no clear divisions between facts and tags, so remember that facts are usually something that people agree on, while tags can be a bit more subjective. In this light, we should probably put genres in the tag category, but that is a matter of debate also.

One of the biggest showstoppers for people trying to use content-based recommenders is that they can't get the data about the items. So, what options would you have? We could try to build it ourselves, or we could hire people to go through the content and tag it. But beware, that can produce strange recommendations. There are places where people tag content for a living, can you guess where?

Let's look at an example where you can get a feel for the differences between tags and facts.

EXAMPLE:

Let's take a film. Opening up at many cinema sites at the time of writing was a commercial on BATMAN V SUPERMAN: DAWN OF JUSTICE (BvS). Those DC comics are really trying their best to compete with Marvel, but that is another story. The site, shown in figure 10.6, is called imdb.com. Looking at the description, you can see that the genres are action and adventure (and sci-fi but I disagree with that) and that the film premiered in 2016. A lot more can be said about the film; for example, it could say that Ben Affleck plays Batman. I would also put that it is a long film.

⁶⁷ Look here for the most popular movies which have been tagged *breaking-the-fourth-wall* <http://www.imdb.com/search/keyword?keywords=breaking-the-fourth-wall>

⁶⁸ <http://www.urbandictionary.com/define.php?term=Breaking%20the%20Fourth%20Wall>

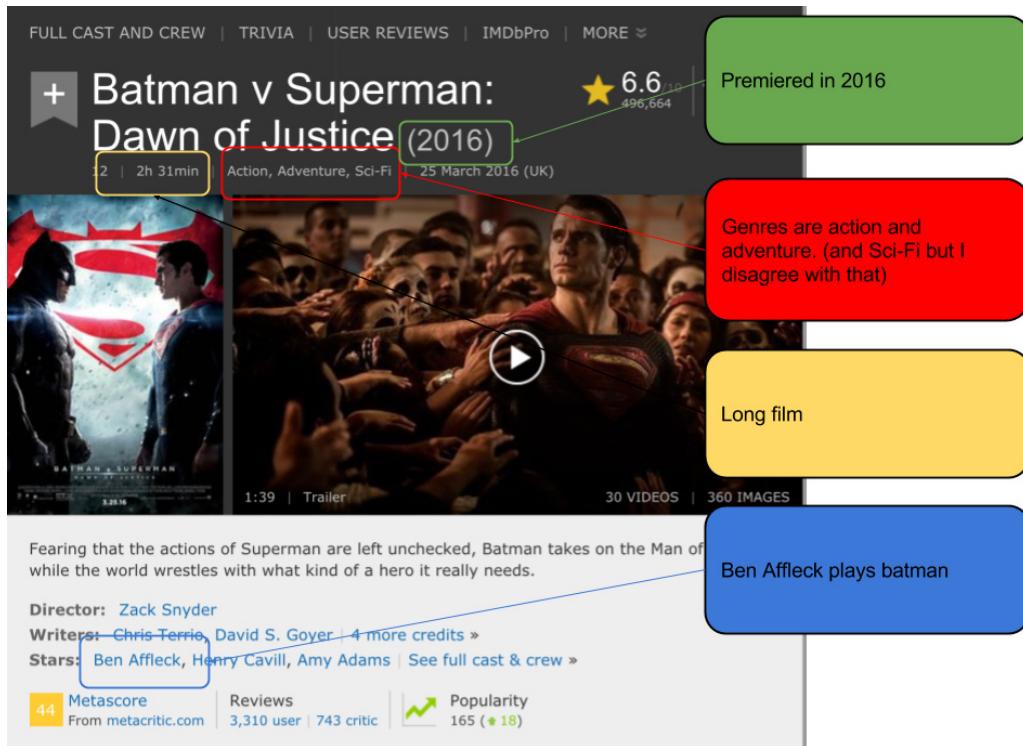


Figure 10.6 Batman vs. Superman film on IMDB.

To represent this, we can make a simple vector like the one shown in Table 10.4. Of course, that is not a vector but a table but think of it as a list of key-values.

There are two types of values:

- **Binary values**

Like *starring Ben Affleck* and *Action*,

- **Quantitative values**

Like *explosions* is a count of how many explosions there were in the film (if that could be considered as something worth having as a feature), and finally, there is the production year.

Table 10.41 An Item profile of Batman vs. Superman

BvS	Year	Starring Ben Affleck	Action	Adventure	Comedy	explosions	Long film	Dogs	Superheroes
	2016	1	1	1	0	5	1	0	1

To be able to show the table/vector here it didn't leave me much space for other features than the ones in Batman vs. Superman, but you should imagine that we want to represent lots of different films using the same vector. So, comedies should fit there too. Films starring popular artists should also be there, maybe a feature indicating how often they smoke, look at the rain, or whether any fish were hurt in the story. In the end, you will have a very long list of different features, and movies described using them. One of the funny parts about doing content-based recommenders is to figure out what is important, and what is not.

Now, if a user bought or liked BvS and other movies starring *Ben Affleck*, we could deduce that the user likes *Ben Affleck*. Then, with that knowledge, we can search the list of vectors for other films with a one in the *Starring Ben Affleck* feature and recommend those. It could of course also be for some other reason that the user bought them, in which case this would not be a good direction to go. That other reason might be that the specific user likes a specific genre of films which is orthogonal on the usual genres, and is therefore not normally used to classify movies. This type of hidden (or should we call it latent) genre is something that we will start looking for as soon as we introduce the LDA.

10.4.2 Categorical data with small numbers

Above we talked about actors, but we can do it even more generally and say everybody worth mentioning who is in the production of the content could be a feature. But while words that only occur only once in a document are good to save if you are creating a search, we need to economize a bit when we talk about recommendations. If there is an actor who only occurs in one film, it's great if somebody likes the film, but it actually doesn't help us in any way to find similar films because this actor isn't found anywhere else. So, we can't use an actor that's only mentioned once to find similar movies. In that case then we might as well leave the actor out.

10.4.3 Converting the year to a comparable feature

Most actors only appear in one role in a film, so usually, the feature that indicates "starring X" is either 0 or 1. Except, of course, if we are talking about Eddie Murphy who typically plays everybody in his films. But would it really make it star Eddie Murphy more if he plays five roles instead of just one? If you like Eddie Murphy, then I guess you like Eddie Murphy. But there are features that we would like to keep as ordinal numbers (things that have an order) so that it's clear one is bigger than another and so on. Production year is a good example. If a user

likes content from 1980, a film from 1981 is probably closer to her taste than one from 2000, so production year is something we'd like to keep ordinal.

When adding an ordinal feature, it is also worth considering how important this feature is. If you put an ordinal feature such as a production year into a system where everything is between zero and one, 2000 is a very high number, even if it is supposed to represent a year. So it is important that you normalize the data or scale it, so it is between zero and one, or close to it.

To illustrate this, we can look at one way you can compare two films by plotting them, as shown in figure 10.6. It is easy to see that even if *Harrison Ford* doesn't star in *Harry Potter*, and they are made (2001 – 1979) 22 years apart, they still look very similar in figure 10.6 (left). While in figure 10.6 (right) the production year has been normalized⁶⁹. You could even subtract 1870 from all the production years or simply the earliest production year to make the difference a bit bigger. With all the data having values between 0 and 1 it is very easy to see that they are different.

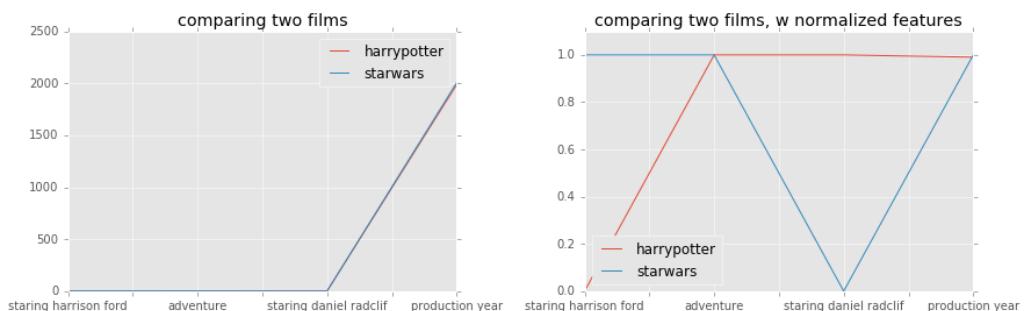


Figure 10.7 Illustrating why data needs to be normalized. Left: production year not normalized and the lines are more or less on top of each other, right: production year divided by the max year and suddenly you can see that the two movies are very different.

Besides the tags and facts described above, we can also use descriptions.

10.5 Extracting Metadata from Descriptions

In the scope of content based filtering, a news article is interesting because firstly they will often only be relevant for a short time, which means that they are hard to recommend using collaborative filtering (you know that algorithm we talked about in the previous chapter). But we still want to recommend them. So besides using popularity, we can try to analyze the

⁶⁹ By dividing each year with the highest year. So 2001 => 1 and 1979 => 1979/2001 = 0.99

actual content. One way of doing that is to look at what words are in the article, how many each occurs and how commonly they appear in all the news items in the database. This can be done using TF – IDF, which we will look at in a bit. An article is text, so the content in the description, while a movie has a description which is written by somebody.

10.5.1 Preparing Descriptions

Getting good descriptions of content is usually not easy. The quality of descriptions can vary a lot. In the movie business, they are fairly good at distributing descriptions, but many others leave much to wish for.

It is still quite a challenge to make computers read and understand a text, at least for this purpose. So before trying to extract information from the descriptions, we need to remove all the things that might confuse the machine. In the following, we will look at exactly that. First, you start out pulling the text apart and put it into a bag.

THE BAG OF WORDS (BOW) AND TOKENIZATION

To use the descriptions, we need first to make a bag of words (BOW), meaning that we will split the description into an array of words

"the man likes big ice creams" -> ["the", "man", "likes", "big", "ice", "creams"]

(notice that using bag of words, we are already losing a bit of information, because "ice" and "cream" mean something when they are right next to each other, while they can express many other things when they are further apart.) This representation will also be the same for any permutation of the words used.

"the big ice man likes creams"

Will produce the same BOW, even if it is talking about something different. There are words that won't add any knowledge to the BOW; these words are referred to as Stop Words.

REMOVING STOP WORDS

Descriptions are full of filler words, in the sense, there are a lot of words needed to make a human-readable description. But since we are deconstructing the documents into an array of word, we lose the value of words like *the* or *a*. Because the word 'a' by itself doesn't give any descriptive information by itself, so it's better to remove words like that. The words you don't want in the model are called stop words.

So the next step is to remove the stop words from the BOW. Stop words are, strangely enough, not synonyms of the word stop, but words that we are going to skip in our analysis, a stop words list is dependent on both your language and your domain. The one we will use here supports English but also many other languages (like Danish). The package can be installed writing the following in the command prompt:

Listing 10.1: Installing the stop words package.

```
pip install stop-words
```

after installing the package, you can get the stop words importing `get_stop_words` as shown here:

Listing 10.2: Importing stop words package.

```
from stop_words import get_stop_words
```

which provides an array of words you probably won't be interested in including into your model.

Calling `get_stop_words('en')` will result in the following: Listing 10.3: English stop words

```
['a', 'about', 'above', 'after', 'again', 'against', 'all', 'am', 'an', 'and', 'any', 'are',
'aren\'t', 'as', 'at', 'be', 'because', 'been', 'before', 'being', 'below', 'between',
'both', 'but', 'by', 'can\'t', 'cannot', 'could', "couldn't", 'did', "didn't", 'do',
'does', "doesn't", 'doing', "don't", 'down', 'during', 'each', 'few', 'for', 'from',
'further', 'had', 'hadn\'t', 'has', "hasn't", 'have', "haven't", 'having', 'he',
"he'd", "he'll", "he's", 'her', 'here', "here's", 'hers', 'herself', 'him', 'himself',
'his', 'how', "how's", 'i', 'i'd", "i'll", "i'm", "i've", 'if', 'in', 'into', 'is',
"isn't", 'it', "it's", 'its', 'itself', "let's", 'me', 'more', 'most', "mustn't",
'my', 'myself', 'no', 'nor', 'not', 'of', 'off', 'on', 'once', 'only', 'or', 'other',
'ought', 'our', 'ours', 'ourselves', 'out', 'over', 'own', 'same', "shan't", 'she',
"she'd", "she'll", "she's", 'should', "shouldn't", 'so', 'some', 'such', 'than',
'that', "that's", 'the', 'their', 'theirs', 'them', 'themselves', 'then', 'there',
"there's", 'these', 'they', "they'd", "they'll", "they're", "they've", 'this',
'those', 'through', 'to', 'too', 'under', 'until', 'up', 'very', 'was', "wasn't",
'we', 'we'd", "we'll", "we're", "we've", 'were', "weren't", 'what', "what's", 'when',
'when's", 'where', "where's", 'which', 'while', 'who', "who's", 'whom', 'why',
'why's", 'with', "won't", 'would', "wouldn't", 'you', "you'd", "you'll", "you're",
"you've", 'your', 'yours', 'yourself', 'yourselves']
```

You would probably want to add some more but to start with this works. Before using the BOW you want to go through each word and check if it is a stop word, if so, remove it.

REMOVING THE HIGH AND LOWS

It is also worth looking at words which appear in all documents, and the ones that only appear few times once. The high-frequency ones will just create background noise, while the low ones will just add complexity to the model, without adding things.

STEMMING

Next, it's worth considering stemming your words, which means that you remove the endings from all words so that words like *run*, *runner* and *running* all become *run*. Its effectiveness depends on a bit on what type of text you are working on. Stemming is useful in long documents but not as much in a list of keywords.

But, before I talk you out of using it, though, let me offer a few more details. Stemming is good in most cases; there are of course words that when you strip them, they become the same but weren't supposed to, but it is generally considered that the good outweighs the bad. The reason why I discourage it on short documents is that stemming does remove some information; if your descriptions are short, it's better to keep words like the are, while with long documents it's a way to avoid data overload.

Before moving on, let's just be sure we know where we are and where we are heading. It should be clear by now what is content filtering and how to do feature extraction. We will now move on to the first of the two ways of extracting features from descriptions and creating something that can be compared by a computer. First, we will look at TF-IDF and the following we will do LDA. These are only two of many different types of feature extractions you can do in text, another one is word2vec.

10.6 Finding important words with Term Frequency – Inverse Document Frequency (TF-IDF)

Finding words with high frequency in few documents

When you are looking at documents for information filtering or search, you often want to look at which words or phrases there are in the documents. But, in addition to the stop words, documents are full of words that are so overused that they don't add anything descriptive to the document. Imagine that you cut this book into small pieces. Each piece would probably contain the word *recommender* zillion times, so even if it is a mega-important word, it wouldn't help to distinguish the documents. If on the other hand you have a collection of articles on the great things of computers, there might only be one article on recommenders and then the word *recommender* would be defining for the article, and probably even more if it is there many times. A simple way to define tf is simply:

$$tf(word,document) = \text{how many times does word appear in document}$$

But more often the following is used

$$tf(word,document) = 1 + \log(word\ frequency)$$

So the more times a word is present in one document, the higher the chance that it is important (assuming that you have removed all the stop words). But as we said above only if it is only present in few documents. For this we use the inverse Document Frequency which is the number of all your documents divided by the number of documents that contains the word. Together the product is tf-idf which is defined like this:

$$tf-idf(word,document) = tf(word,document) * idf(word,document)$$

For example, if we look at the following texts (let's pretend each line of text is a document):

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/practical-recommender-systems>

Licensed to Asif Qamar <asif@asifqamar.com>

"The superhero Deadpool has accelerated healing powers"
 "The superhero Batman takes on the superhero Superman"
 "The LEGO superhero Batman takes the stage"
 "The LEGO superheroes adventures"
 "The LEGO Hulk"

If we want to calculate the importance of a superhero, we can use the following calculation:

Table 10.3 Showing term frequency and inverse document frequency of the word "superhero". We are not using stemming so "superheros" (in doc 4) does not work.

		Term Frequency "superhero"	Inverse Document Frequency "superhero"	Tf*Idf
1	"The superhero Deadpool has accelerated healing powers"	1	5/3	1.66
2	"The superhero Batman takes on the superhero Superman"	2	5/3	3.33
3	"The LEGO superhero Batman takes the stage"	1	5/3	1.66
4	The LEGO superheroes adventures	0	5/3	0
5	The LEGO Hulk	0	5/3	0

If we then have somebody looking for a superhero movie, then we can show the second document ("The superhero Batman takes on the superhero Superman "), which has the tf-idf("superhero") equal to 3.33, followed by the two other documents with non-zero values. Try on your own to calculate tf-idf("Hulk")⁷⁰

So, why all this talk about words when we were talking about features of content? Well, when we have found words that provide large tf-idfs, we can add those to the list of our features. So if the sentences above were actual descriptions of movies, we could add "superhero" to the list of features with the value of the tf-idf.

The formulas above were very simple. Usually you'd use the following idf formula:

$$idf(term) = \log \frac{\text{Total number of docs}}{\text{Number of docs containing term}}$$

⁷⁰ $1(5/1)=5$

This is used because it will keep the final number more stable. Since the log of numbers between 1-10 are quite close to each other. TF-IDF was once king, but after the invention of LDA models it has fallen out of grace, and everybody's first choice is now to use LDA models, or similar topic models. However, before you believe I just took away time from you to explain something you might not use, it's worth mentioning that TF-IDF is something that you can use to clean the input of the LDA. TF-IDF is quite a classic and have been widely used, but new algorithms are also gaining traction, one particularly called Okapi BM25⁷¹.

10.7 Topic modeling using the Latent Dirichlet Allocation (LDA)

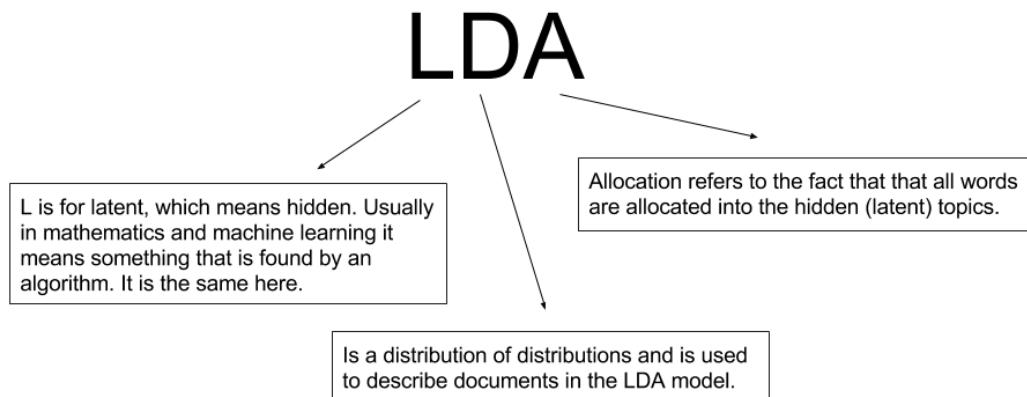


Figure 10.8 LDA is short for Latent Dirichlet Allocation

If you are a machine learning professional, you've probably heard a lot about the fantastic LDA models (figure 10.8), which can solve everything regarding text. And the way that they are often described, it also sounds like they will give you a foot massages and wash your windows!

If you are not, then sharpen your pencil and stay alert, because it is a bit complicated. Luckily LDAs are quite easy to use, in the sense that there are many libraries out there that make it easy to have some code up and running in no time. And in no time, you will start having trouble. Because even if it did give you a foot massages, it really doesn't respond in an easy language.

Let's go through a bit of the theory of how it works, and then return to what it produces. An LDA is what you call a generative model. So, let's start out with an example explaining what that is.

⁷¹ https://en.wikipedia.org/wiki/Okapi_BM25

GENERATIVE MODEL EXAMPLE

Let's start out with a silly example that hopefully will make it more clear. If we had 3 bags each with shapes of one color - one bag with red shapes, one with blue shapes, and one with green shapes. If you had to form the row of shapes shown in figure 10.8, you would need to draw three times from the red bag, two times from the blue one, and finally five times from the green one.



Figure 10.9 row of shapes

So, we could say we could generate this row of shapes by drawing the sequence above, to form the line in figure 10.8 we would have to draw a shape from the red bag 30% of the times, 20% from the blue bag and the remaining 50% from the green or in other words, we could say that the row X above is formed by the following equation:

$$x = 0.3*\text{red} + 0.2*\text{blue} + 0.5*\text{green}.$$

If we said that this was a recipe for creating documents, then we could also have created the document shown in figure 10.8



Figure 10.10 another row, which could have been generated using the following recipe $0.3*\text{red} + 0.2*\text{blue} + 0.5*\text{green}$.

Now look at another row (figure 10.9), this one can be defined as $x=0.2*\text{red} + 0.3*\text{blue} + 0.5*\text{green}$.



Figure 10.11 another list of shapes which could be generated using the following recipe $0.3*\text{red} + 0.3*\text{blue} + 0.5*\text{green}$

The only difference between the first and the third rows is that a red triangle is exchanged by a blue box. If we imagine each of these shapes are pieces of description, then the idea is that we can write them down a recipe or formula down as recipes will make it easier to compare.

Try to keep this mental imagine of the bags with shapes that can be used to generate rows of shapes.

What if instead of bags, we call them topics, the shapes are words, and the rows are descriptions. Then instead of colors the topics could be something like *Superhero*, *Computer Science* and *Food*, and inside that topic of *Superhero* it could contains words like *Spiderman*, *flying*, *strong*, *superhero* etc. like the following:

Superhero: Spiderman, flying, strong, superhero

Computer Science: Computer, laptop, CPU

Food: eat, breakfast, fork

If we have a description like the following:

Z = Spiderman is home with his laptop to eat breakfast with a fork.

If we wanted to generate that from our three topics we can draw *Spiderman* and *flying* from the *Superhero* topic, *laptop* from *Computer Science* topic and finally *eat*, *breakfast* and *fork* from the *Food* topic or, in other words, we can say the description is:

$$Z = 0.2 * \text{Superhero} + 0.1 * \text{ComputerScience} + 0.3 * \text{Food}$$

That's quite simple, isn't it? It does get a bit more complicated because each word in the topic has a probability which says something about how important a word it is. But let's just keep that in mind for now, and skip on to how we create these topics. An example a bit closer to our MovieGEEKs site can be seen in figure 10.11, which shows how topics related to movie genres could be distributed.

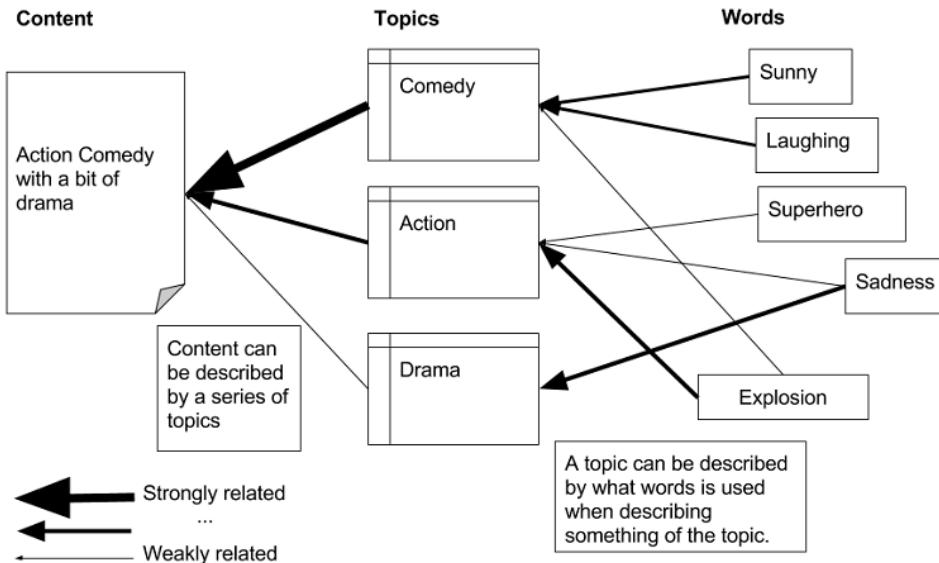


Figure 10.12 A topic model means a list of topics; each topic is defined by a list of words and their respective probability of being drawn. A document can be described using topics, using a recipe of how big a percentage of times you should draw from each topic.

I came up with the topics above, but normally the whole idea of topic models is that you want the computer to sort out topics for your database of descriptions.

Generating the topics

So, how do you calculate them? The contract, as shown in figure 10.13, is that input of the LDA algorithm is a bunch of documents and a number K. The documents are the texts we have talked a lot about, and the K is the number of topics the algorithm should create. The output of the algorithm is a list of K topics, and a list of vectors one for each document, containing a probability for each topic being in the document. Topics can be generated in several different ways, we will look at Gibbs Sampling.

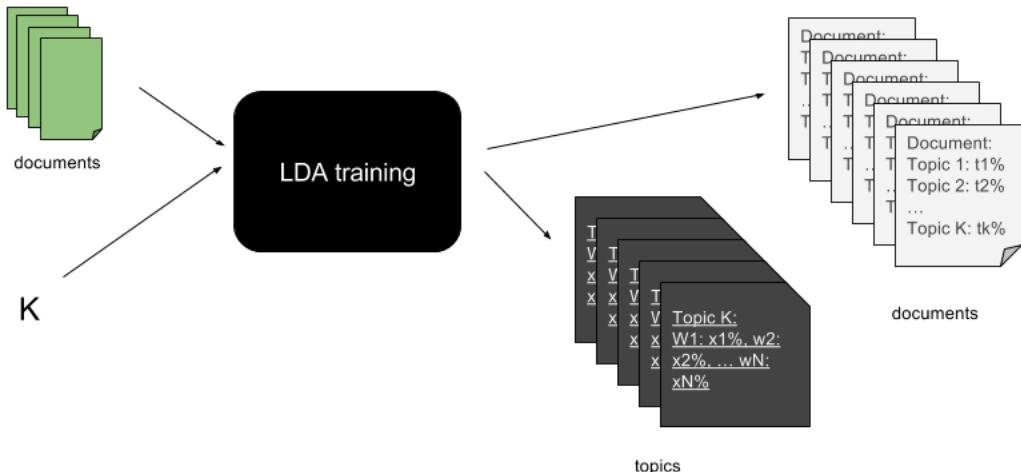


Figure 10.13 When running an LDA algorithm the input should be a list of documents and the output of training an LDA model in the shape of topics, each containing a list of words, where each word has a probability in the topic.

GIBBS SAMPLING

Looking again at Figure Figure 10.13, we can elaborate a bit on what data structures we have. We have the K , which is the number of topics the LDA model should contain, and then we have each document is essentially considered a bag of words (which means that we don't have any structure, or idea which words are next to each other, only the words).

The goal is now to connect words with topics and documents with topics.

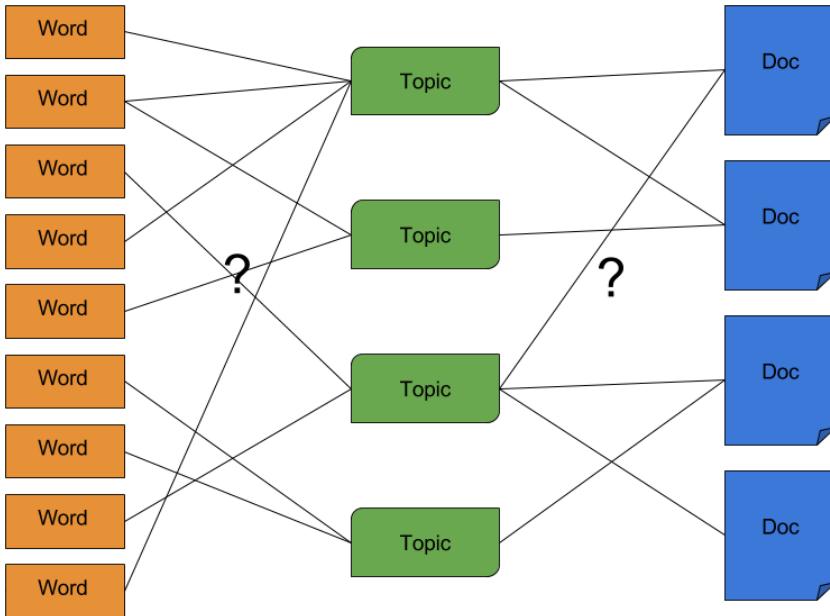


Figure 10.14 The goal is to connect the words and docs with the topics.

Start out with the topics and the words for example as shown in figure 10.14. If you line up a list of K topics and all the words which are present in the documents, it's hard to imagine ever to come up with a solution. But since words related to the same thing are usually found in the same documents, then you already have some information to go with.

Words that are in the same topic are often found in the same documents.

This means that if you know, one word is already in a topic, then you have some information about whether another word should be there.

This is what the Gibbs sampling takes advantages of. It starts out by randomly placing all words in topics, then it will try to adjust one word according to the rules we know about which words appear together. Now it will go through each word for each document and construct a weight for the specific word is actually in the chosen topic, using knowledge of the current distribution of topics in the document and distribution of words in the topic. By fitting each word and distribution one step at the time, The Gibbs sampling algorithm, will slowly approach a distribution of words and topics that seems magically to make sense. To understand the

magic of Gibbs sampling better have a look at the following footnote⁷², it goes into all the nitty details I couldn't fit here. I don't know if you will need this information, but I believe that you need to know how a model is trained and Gibbs sampling is used to train the LDA model, so here you go.

LDA MODEL

The Gibbs sampling will produce K topics which will look something like the following printout if K was ten it could look like the following. Each topic is printed with the ten most probable words in each topic. There are some strange words here also like *18genre* which is my attempt to inject genres into the description also. Instead of using names like *action* and *drama* I added numbers to be sure they are not confused with words in the description, and thereby changes the importance of some of those words.

Listing 10.4: Topic distribution

```
[(),  
  '0.037*18genre + 0.022*love + 0.021*s + 0.012*young + 0.012*man + 0.011*story + 0.009*woman  
  + 0.008*father + 0.008*35genre + 0.007*10749genre'),  
(1,  
  '0.010*vs + 0.010*even + 0.009*nothing + 0.008*based + 0.007*boyfriend + 0.005*s +  
  0.005*music + 0.005*rise + 0.004*writer + 0.004*hero'),  
(2,  
  '0.012*one + 0.011*s + 0.011*gets + 0.011*three + 0.009*like + 0.008*99genre + 0.008*years  
  + 0.007*car + 0.006*different + 0.006*marriage'),  
(3,  
  '0.013*s + 0.010*28genre + 0.009*house + 0.009*two + 0.009*878genre + 0.008*years +  
  0.008*daughter + 0.007*year + 0.007*world + 0.007*old'),  
(4,  
  '0.024*35genre + 0.015*girl + 0.011*10749genre + 0.010*year + 0.009*old + 0.009*go +  
  0.009*falls + 0.008*s + 0.008*get + 0.008*four'),  
(5,  
  '0.025*53genre + 0.016*80genre + 0.015*27genre + 0.013*28genre + 0.011*18genre +  
  0.008*polic + 0.008*s + 0.007*goes + 0.007*couple + 0.007*discovers'),  
(6,  
  '0.016*t + 0.016*s + 0.011*school + 0.010*friends + 0.010*new + 0.009*boy + 0.008*first +  
  0.007*will + 0.007*35genre + 0.007*show'),  
(7,  
  '0.013*99genre + 0.008*s + 0.008*fall + 0.007*movie + 0.007*documentary + 0.007*power +  
  0.005*18genre + 0.005*wants + 0.005*will + 0.005*move'),  
(8,  
  '0.046*film + 0.011*friend + 0.010*past + 0.009*18genre + 0.009*directed + 0.007*upcoming +  
  0.007*produced + 0.006*ways + 0.006*festival + 0.006*turned'),  
(9,  
  '0.016*s + 0.014*will + 0.013*one + 0.010*10402genre + 0.009*life + 0.009*journey +  
  0.008*game + 0.007*work + 0.007*time + 0.006*world')]
```

⁷² Gibbs sampling for the uninitiated: <https://www.umiacs.umd.edu/~resnik/pubs/LAMP-TR-153.pdf>

Each topic actually contains all words found all the documents that were used, only many words have so little probability (close to zero) that they are not interesting. The probabilities are the numbers in front of each word in the topic listing in Listing 10.4.

The number in front of each word is the probability that if you try to generate a document containing that topic then you will draw that word with that probability. So if you have a document that was generated from topic 0 alone, then each word has the probability of 2.2% of being love. As you can see in the following topic snippet:

Listing 10.5: Topic 0

```
[ (0,
  '0.037*18genre + 0.022*love + 0.021*s + 0.012*young + 0.012*man + 0.011*story + 0.009*woman
  + 0.008*father + 0.008*35genre + 0.007*10749genre'),
```

If we look at a document instead, the model will represent a document like the following:

Listing 10.6: A documents topic distribution

```
[ (2, 0.02075648883076852),
  (3, 0.1812829334788339),
  (9, 0.78545976997831202)]
```

This means that the selected document could be generated by topics 2, 3, and 9. Meaning the document can be generated by pulling words from topic bags 2, 3 and 9. 2.1 % of the time a word is generated from topic 2, while 18.1% from 3, and finally 78.5 % from 9.

So, know you know how it works, in theory at least. A lot of the work in making LDA function is how you process the text you are inputting. It is hard to teach how to understand LDA models. You need to stare at the output for a bit. Well, actually more like for a long time before anything makes sense. Because the topics might be representing something that is hard to understand from the input.

THE CORPUS

So far we only talked about adding descriptions and documents into the LDA, but what kind of documents are they?

The example above and the implementation in MovieGEEKs that we will soon talk about, both use the same documents that we want to calculate similarity between. But it is a good idea to use a dataset which describes or at least are good examples of some of the topics you want to find.

Issuu.com uses an LDA model to give you recommendations, among others. They use The Wikipedia dataset to create the models, which enables them to make a model with quite clear topics. Using Wikipedia not only makes the topics easier to understand but since it contains documents with classifications for everything, it will ensure that the topic model will be nicely distributed.

ADDING FEATURES AND TAGS TO DOCUMENTS

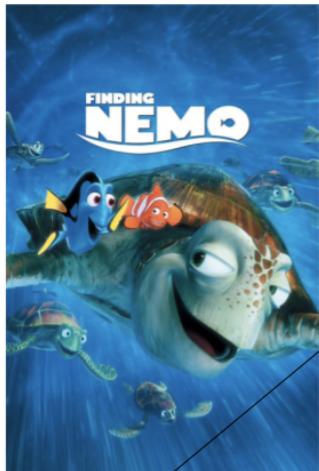
A document is presented as a bag of words, and not as connected words where the order is important. It means that if we want to control the topics, a document will contain, we can also inject words into the descriptions. For example, if we want to include information like actors or product year, then we simply add the words to the bag of words (BOW). You can see an example in the topic model above where words like '35genre' appeared, but more detail on that in a bit.

10.7.1 What knobs can we turn to tweak the LDA?

So, all this magic, does it work? Well if you look at figure 10.15 then it's easy to see that it leaves something to wish for.

A model is only as good as the quality of the documents. Good quality documents will be good examples of the different types/genres/subjects that you want to model to represent. For example, the descriptions of movies that we are using are not always good for this. Looking at the *Finding Nemo* description it doesn't really say anywhere that it is a cartoon, or it's about the sea. We can add the genres also that will include that it is an animation, but again unless we want to start handwriting better descriptions, it's hard to come up with ways to deal with it.

Finding Nemo



Released:
2003-05-30

Description:
Nemo, an adventurous young clownfish, is unexpectedly taken from his Great Barrier Reef home to a dentist's office aquarium. It's up to his worrisome father Marlin and a friendly but forgetful fish Dory to bring Nemo home — meeting vegetarian sharks, surfer dude turtles, hypnotic jellyfish, hungry seagulls, and more along the way.

Language:
en

Average rating:
7.6

Genres:
[Adventure | Comedy | Animation]

Buy

Finding the sequel to this film is a good sign, and so is *All Dogs go to Heaven 2*, and the minions film.

You could discuss whether finding *Frankenweenie* is good, but it is a story where something is brought back, in Nemo is back home, in *Frankenweenie* it is back to life, it is a Disney film and rated U so its not too bad.

Innocenza e turbamento is quite a long way from *Finding Nemo*, but again if you read the description it is about coming home

Similar content



Figure 10.15 content based recs from LDA which could need a bit of tuning, a Disney cartoon which is similar to *Innocenza e turbamento* doesn't sound like the best of recommendations.

But a check you can do is to see how the recommender responds to something you know have many similar items in the catalogue. Something like spider-man movies, as shown in figure 10.16

User: 400004

Movie GEEKs

Genres

- Comedy
- Horror
- Drama
- Fantasy
- Biography
- Thriller
- Film-Noir
- Talk-Show
- Crime
- Western
- Music
- Musical
- War
- Reality-TV
- Sport
- Game-Show
- Short
- History
- News
- Adventure
- Sci-Fi

Spider-Man: Homecoming



Released:
2017-07-05

Description:
Following the events of Captain America: Civil War, Peter Parker, with the help of his mentor Tony Stark, must balance his life as an ordinary high school student in Queens, New York City, with fighting crime as superhero alter ego Spider-Man as a new threat, Vulture, emerges.

Language:
en

Average rating:
7.3

Genres:
| Adventure | Action | Fantasy |

Buy

A good test on whether similarity works in this dataset is that the Content based recommender returns all spider-man films when the spider man film is a seed.

Similar content



Figure 10.16 The content-based recommender understands that all the different Spider-man movies are similar

Besides the input, it is also worth to consider the number of topics you use.

WHAT IS A GOOD NUMBER OF TOPICS?

One thing that is make or break for the LDA model is whether or not you have trained it with a good number of topics. And it is also one of the things that makes it into an art of doing LDAs because there isn't really any way to verify that you have the right number. In the analytics part of the MovieGEEKs site you have the following interactive visualization of the model (<http://127.0.0.1:8001/analytics/lda>) and shown in figure 10.17. It shows a circle for each topic. Hovering over a topic you get a list of the most frequent words in that topic. It is important to try to find a number of topics that are distributed as good as possible. That doesn't mean too little but neither too much, if you select a too little K then more documents

will look like each other, like if you have to describe all films using only the genres Action, Comedy or Drama. But if you select too many topics, then you might end up with no documents ever similar because you have a million different dimensions (topics); then we are back to the curse of dimensionality.

Lda model (10 topics)

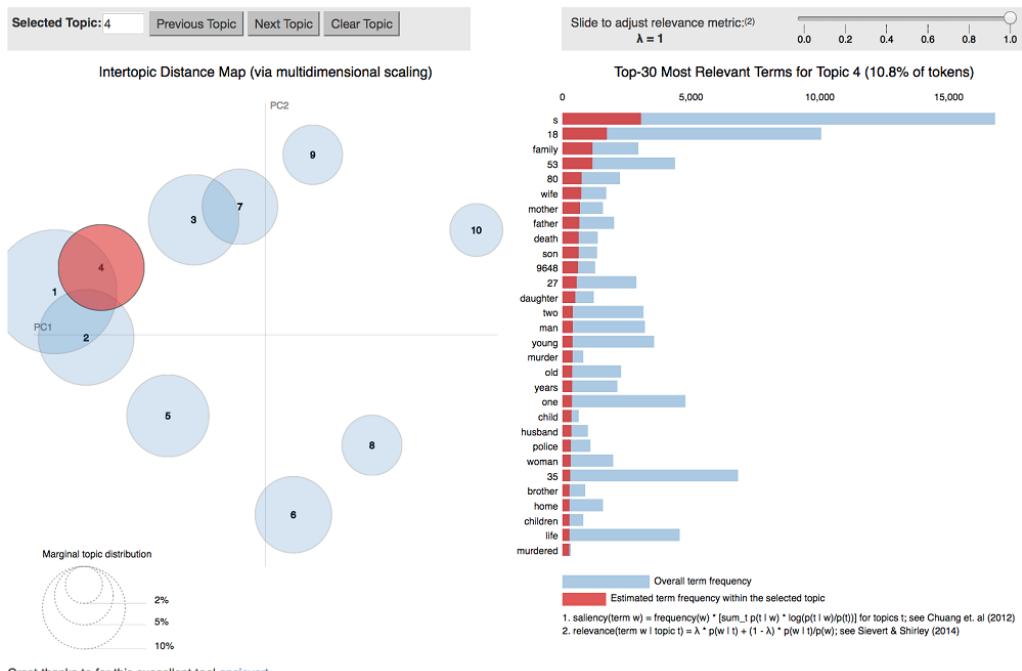


Figure 10.17 pyLDAvis dashboard

The dashboard is created using something called pyLDAvis⁷³, which is described in a paper⁷⁴.

It is a good idea to test out the following when deciding on the right number:

- Check a view like the one shown in Figure 10.17(or similar) to see if the topics are distributed and not on top of each other.
- Test if the LDA produces similar items, further on in this chapter you will see how LDA is implemented in the MovieGEEKs site, and where you can check if it produces good recs. The first attempt produced the recs shown in Figure 10.15.

⁷³ <https://pyldavis.readthedocs.io/en/latest/>

⁷⁴ <http://nlp.stanford.edu/events/illvi2014/papers/sievert-illvi2014.pdf>

The short answer to how it should work is that it should look right. So, find some movie that you like and try to tweak it until that makes sense. Then ask a friend to do the same, and see if you can tweak it, so you are both happy.

PLAYING WITH ALPHA AND BETA

There are two more parameters to play with when you are doing LDA. It quickly becomes hairy but overall you can change alpha, and beta, to change the distributions of both the documents and the words in the topics.

If you put a high alpha, then you will distribute each document over many topics, while low alpha only a few topics. The advantage with the high alpha is that documents seems to be more similar, while if you have very specialized documents, then a low alpha will keep them divided into few topics.

The same is true for beta: a high beta will lead to topics more similar since the probabilities will be distributed on more words will be used to describe each topic, so instead of having 10 words in a topic with probability above 1% you might have 50, this allows for bigger overlap.

Most people I have talked with are very careful to change the default values of alpha and beta, but if there is time to play, you should.

10.8 Finding similar content

Now that we have the LDA model, we have another way of finding similar items. Simply by projecting two documents into the LDA model, we can calculate the similarity of the two. Since the probability distribution can be seen as vectors, many have simply used one of the similarity functions that we have talked about in chapter seven, for example, cosine similarity.

In principle, it is also possible to use the LDA model to compare documents that were not used when creating the model, this is one of the more important features for content-based recommenders, to get around the problem of cold products, which we talked about back in chapter 6.

Later in this chapter, we will talk more about the actual implementation in MovieGEEKs and show a similarity calculation between two movies. This can be used for the *more like this* recommendation. Now let's have a look at how to do personalized recommendations in a content-based recommender system.

10.9 Creating the user profile

But if you like James Bond, then you might also like ...

That is often how it goes, you talk about films with a friend, and the above sentence goes.

If we wanted to create a function providing this in our system, then we would create something that took a list of items that the user likes, and returned some other items which the user might also like.

In real life, such a function is individual since the user might not be uniquely defined by the items consumed (even if most recommender systems probably will respond the same), in reality, it also depends on many other things. But let's start out with this simple example.

What we can do is to just iterate through each of the items the user likes and for each find similar items. When you got a list, order it according to similarity and the users ratings from the original list of consumed items. More formally we could do the following for an active user:

1. Get all consumed items CI by the active user
2. For each item I in CI do
 - a. Find similar objects using the LDA
 - b. Calculate a rating based on the similarity and the active users rating.
3. Order items by rating.
4. Order by relevancy

10.9.1 Creating the user profile with TD-IDF

With the vectors that we described above regarding the tags and facts there is another way you could create a user profile simply aggregating the vectors of the things the user likes and subtracting the things the user doesn't like. If we look at the insight full table 10.3

Table 10.54 vector representation of some movies

	Starring Ben Affleck	Action	Adventure	Comedy	explosions ⁷⁵
BvS	1	1	1	0	5
Valentines's day	1	0	0	1	0
Raiders of the lost ark	0	1	1	1	2
La La Land	0	0	0	1	0

If we now have some a user (we are not doing collaborative filtering here, so it is enough just to look at one), who has rated *Raiders of the lost ark* 5 stars (who doesn't love that movie?) and La La Land 3 stars. We could now create a user profile adding multiplying the rating on the movie vector and adding it together. As shown in table 10.4

⁷⁵ These are made up numbers.

Table 10.5 Multiplying the users ratings on the movie vectors and adding each element to create the user profile.

	Starring Ben Affleck	Action	Adventure	Comedy	explosions ⁷⁶
Raiders of the lost ark	0	5	5	5	10
La La Land	0	0	0	3	0
Users profile	0	5	5	8	10

We can now use that vector to find similar content which are similar to the users taste. We should probably normalize the values such that they are on the same scale as for the movies. Actually the number of explosions should probably be downplayed a bit, into a film that contains explosions or not. In which case the user profile would look like the following:

Table 10.6 Multiplying the users ratings on the movie vectors and adding each element to create the user profile.

	Starring Ben Affleck	Action	Adventure	Comedy	explosions ⁷⁷
Users profile	0	5	5	8	5

I find this a bit more true to the films. Now good thing with this is that we will be looking for movies that has all the aspects of what a user likes (that is if we captured the right tags and facts) but the thing is that I like chocolate and lasagna but that doesn't mean that I like them together. The same can be said for many other attributes we could come up with either food or movies. So, the take away is that you could use this to see that the user likes comedy more than action and adventure.

In the MovieGEEKs analytics part of the system I have represented a users taste simply by running through the users rated movies and do sums of the ratings for each genre, like shown in the following code:

Listening 10.7: Extracting taste from ratings. Snippet from analytics/views.py

```

for movie in movies:
    id = movie.movie_id
    r = ratings[id].rating
    for genre in movie.genres.all():
        if genre.name in genres.keys():
            genres[genre.name] += r
    
```

⁷⁶ These are made up numbers.

⁷⁷ These are made up numbers.

```

max_value = max(genres.values())
max_value = max(max_value,1)          ④
                                     ⑤

genres = {key: value / max_value for key, value in genres.items()}      ⑥

```

- ① for each movie the user has rated.
- ② get the rating.
- ③ iterate over each genre for the movie, building a dictionary with genre names as keys and sum of ratings as value.
- ④ Find the max values
- ⑤ Make sure it is not below one.
- ⑥ normalize the values.

This code is used to create taste charts like the following, for user 40006 (<http://localhost:8001/analytics/user/400006/>)

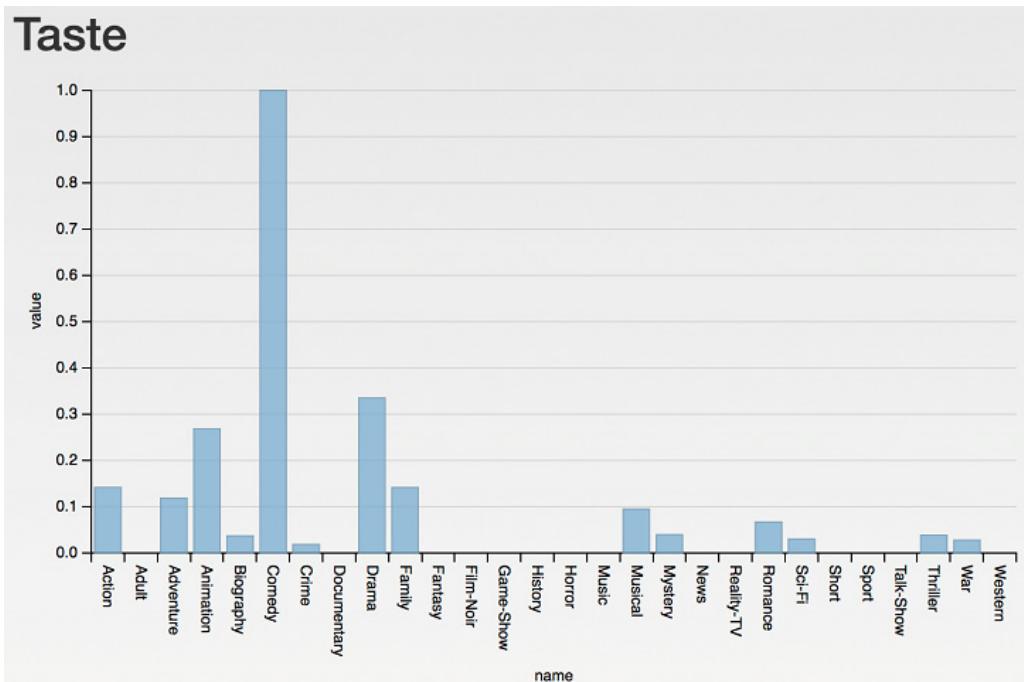


Figure 10.18 taste of user 40006

10.10 Content-based recommendations in MovieGEEKs

As mentioned several times in the chapter, we will go through an implementation and usage of a content-based LDA model building. First, we need to say two words about getting the data.

10.10.1 Loading data

The data set we are using does not contain descriptions of movies, so we are again at the mercy of the good themoviedb.org to retrieve data. In the root of the code following the book, there is a script called `populate_sample_of_descriptions.py` which retrieves the description of the most recent films. The script “only” gets around 1000, so it is not a big dataset, but since I am using the free website themoviedb.org it wouldn’t be fair to get more. An example of what is downloaded can be seen in figure 10.12

```
- {
  poster_path: "/z09QAf8WbZncbitewNk6lKYMZhsh.jpg",
  adult: false,
  overview: '"Finding Dory" reunites Dory with friends Nemo and Marlin on a search for answers about her past. What can she remember? Who are her parents? And where did she learn to speak Whale?"',
  release_date: "2016-06-16",
  - genre_ids: [
    16,
    10751
  ],
  id: 127380,
  original_title: "Finding Dory",
  original_language: "en",
  title: "Finding Dory",
  backdrop_path: "/iWRKYHTFlsrxFtfQqFOQyceL83P.jpg",
  popularity: 27.117383,
  vote_count: 1234,
  video: false,
  vote_average: 6.69
},
```

Figure 10.19: The json object representing *Finding Dory*

The following script shows how to retrieve and save the descriptions to the database. It is shown because you might want to fiddle a bit with it, to make it work for your current setup.

Listing 10.8: `get_descriptions` method in `prs/moviegeek/populate_sample_of_descriptions.py`

```
def get_descriptions():
    url =
        """https://api.themoviedb.org/3/discover/movie?primary_release_date.gte={}&api_key={}
    page={}"""\n    ①
    api_key = get_api_key()

    for page in range(1, NUMBER_OF_PAGES):  
        ②
        r = requests.get(url.format(start_date, api_key, page))
        for film in r.json()['results']:
            ③
            id = film['id']
            md = MovieDescriptions.objects.get_or_create(movie_id=id)[0]
            md.imdb_id = get_imdb_id(id)
            md.title = film['title']
            md.description = film['overview']
            md.genres = film['genre_ids']
```

```

    md.save()
    print("{}: {}".format(page, r.json()))

```

④

- ① url to the movie api being used, films produced after 1970 up til today
- ② run through the all the pages of the search result.
- ③ run through all films in the page
- ④ save to db.

The url contains two dates one says that the release date has to be (`gte`) greater or equal than this and another one says that the release date should be less than or equal(`lte`). To retrieve these movie descriptions run the following:

Listing 10.9: Download the movie descriptions

```
$ python populate_sample_of_descriptions.py.
```

This will download the descriptions to be used for the topic model. There might be a bit of issues running it as the descriptions are collected from www.themoviedb.org, and they have a limit on how many requests you can do, I have added a `time.sleep(1)` in the loop that requests for it to sleep 1 sec between each request. It might not be enough, so if you keep getting angry messages from the server go to the `populate_sample_of_descriptions.py` file and put a higher number.

10.10.2 Train the model

We will be using a library called Gensim, which contains an implementation of LDA models, there are others out there but this is one of the most popular, and my experience with it so far has been good.

You can install Gensim running the following command, it is part of the general requirements of the MovieGEEK site, so if you have installed those, then you should be good to go (besides running `pip3 install -r requirements.txt`):

Listing 10.10: Download Gensim package⁷⁸

```
$ pip3 install gensim.
```

Using the Gensim library it is not too hard to create the LDA model. We simply need the documents. It is done as follows:

Listing 10.11: Builder\LdaBuilder.py - Building a LDA model

```
texts = []
```

⁷⁸ <http://radimrehurek.com/gensim/install.html>

```

tokenizer = RegexpTokenizer(r'\w+')
for d in data:
    raw = d.lower()                                1
    raw = d.lower()                                2

    tokens = tokenizer.tokenize(raw)
    stopped_tokens = self.remove_stopwords(tokens)  3
    stopped_tokens = stopped_tokens                4

    stemmed_tokens = stemming(stopped_tokens)       5

    texts.append(stemmed_tokens)                   6

dictionary = corpora.Dictionary(texts)            7

corpus = [dictionary.doc2bow(text) for text in texts]  8

lda_model = models.ldamodel.LdaModel(corpus=corpus, id2word=dictionary,
                                      num_topics=n_topics)  9

index = similarities.MatrixSimilarity(corpus)     10

self.save_lda_model(lda_model, corpus, dictionary) 11
self.save_similarities(index, docs)               12

```

- 1 go through each document.
- 2 make all the capital letters in the text small letters.
- 3 split the text into an array
- 4 remove the stopwords
- 5 this line is where you would stem the words
- 6 add the concentrated document
- 7 now create a bag of words
- 8 create the corpus, which contains an array of all the documents represented as bag of words
- 9 the lda model being build.
- 10 Create a similarity matrix
- 11 Save the LDA model
- 12 Save the similarities in the database.

10.10.3 Creating item profiles

The item profiles are created with the model and are presented using the LDA vector, which we have talked about before. In the builder we saw above, I have chosen to go one step further and create the similarity matrix directly, such that doing the recommendations is just a matter of looking up the item in question. This is not always possible as you probably have many more items than we used here.

It's a good idea to look at how big the similarities are in the similarity matrix, if all are close to zero then it might be worth trying to come up with more general data which will connect more items.

To create the model and thereby enable MovieGEEKs site to produce content based recs, you simply need to run the LdaBuild.py file by

Listing 10.12: Run the model builder

```
$ python -m builder.lda_model_calculator
```

10.10.4 Creating user profiles

I guess I said it several times, a user profile can be created in many ways, the simplest one, which we will use here, is just a list of the items which the user liked, and for each of them find similar items.

- 1 get all the ratings from the user, at least the 100 highest
- 2 call the item to item recommender method
- 3 run through all the movie ids.
- 4 G a description for the current movie.
- 5 look up the Lda vector in the corpus
- 6 retrieve the similarities
- 7 sort the similar items
- 8 find the movie items corresponding to the Lda vectors
- 9 run through the movies and add them to content_sims if they are not present already or if their sim is larger, than the previously stored version of the movie.

⑪ sort and return the data. Listing 10.14: recs/content_based_recommender.py Doing the recommendation for a user at runtime

```
def recommend_items(self,
                    user_id,
                    num=6):

    movie_ids = Rating.objects.filter(user_id=user_id)
        .order_by('-rating')
        .values_list('movie_id', flat=True)[:100] ①

    return self.recommend_items_from_items(movie_ids, num) ②

def recommend_items_by_ratings(self,
                               user_id,
                               active_user_items,
                               num=6): ③
    content_sims = dict()

    movie_ids = {movie['movie_id']: movie['rating'] \ ④
                for movie in active_user_items}
    user_mean = sum(movie_ids.values()) / len(movie_ids) ⑤

    sims = LdaSimilarity.objects.filter(Q(source__in=movie_ids.keys())
                                         & ~Q(target__in=movie_ids.keys())
                                         & Q(similarity__gt=self.min_sim)) ⑥
    sims = sims.order_by('-similarity')[:self.max_candidates] ⑦
    recs = dict()
    targets = set(s.target for s in sims)
    for target in targets: ⑧
        pre = 0
        sim_sum = 0

        rated_items = [i for i in sims if i.target == target] ⑩
        if len(rated_items) > 0: ⑪
```

```

    for sim_item in rated_items:
        r = Decimal(movie_ids[sim_item.source] - user_mean)
        pre += sim_item.similarity * r
        sim_sum += sim_item.similarity

    if sim_sum > 0:
        recs[target] = \
            { 'prediction': Decimal(user_mean) + pre/sim_sum,
              'sim_items': [r.source for r in rated_items]}

return sorted(recs.items(),
              key=lambda item: -float(item[1]['prediction']))[:num]

```

- 1 we use all the ratings from the user, at least the 100 highest.
- 2 call the recommend_items_by_ratings.
- 3 Having this method makes it easier to test, since it receives a list of what ratings the user got.
- 4 Extract all the id's of the rated movies.
- 5 Calculate the user mean.
- 6 Look up all the lda similarities where the user's ratings are the source, and not the target. Filter on min similarity.
- 7 Order the similarities by similarity value and take the top candidates. Number to take is something you should adjust, balancing performance and time.
- 8 We are only interested in iterating over the targets of the similarity. So, get those.
- 9 For each target we predict a rating to see if it could be a candidate to be recommended, using the users other ratings.
- 10 Find all the items which the user rated that is similar to the current target.
- 11 ,
- 12 If we found rated items. Run through it
- 13 Deduct the user mean from the rating.
- 14 Multiply similarity with rating.
- 15 Add the similarity to the sum.
- 16 If there are any similarities.
- 17 Add it as a recommendation.
- 18 Return a sorted list of recs. Ordered by prediction.

This method can be called using the api, calling <http://127.0.0.1:8000/rec/cb/user/3/> if you want to get recs for user id 3.

The result will look like the following:

Listing 10.14: Result of calling the cb user rec

```
{
  user_id: "400003",
  data:[
    ["1049413",{
      prediction: "10.0000",
      sim_items:[ "2709768"]}
    ],
    ["4160704",{
      prediction: "6.8300",
      sim_items: [ "1878870" ]}
    ],
    ["1178665",{
      prediction: "6.8300",

```

```
sim_items: ["1878870"]}  
], ...
```

The predictions are done based on the rating of only one film. You can discuss whether it is fair to do rating predictions this way, but it is at least an ordering which will provide you with a list where the items that are more similar to the items the user has rated higher.

10.10.5 Showing recs

The recommendations are shown on the details page on the site and is simply an ordered list of items which are similar to the current item. We saw an example of this in figure 10.15

For the personalized recommendations on the front page, we will simply iterate through all the user's items and calculate similarity based on their LDA vectors, and then order them before returning them to the user. An example that shows this can be seen in figure 10.19

User: 400004

Movie GEEKs

Search

Genres

- Comedy
- Horror
- Drama
- Fantasy
- Biography
- Thriller
- Film-Noir
- Talk-Show
- Crime
- Western
- Music
- Musical
- War
- Reality-TV
- Sport
- Game-Show
- Short
- History
- News
- Adventure
- Sci-Fi
- Adult
- Action
- Documentary
- Romance
- Animation
- Mystery







Based on what you and others have bought



Users who liked what you like also like



Similar Content

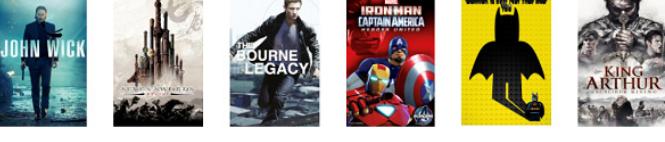


Figure 10.20: The personalized content-based recommendations can be seen on the front page in the row called Similar Content

10.11 Evaluation of the content-based recommender

Before moving on, let's look back at chapter 9 a second and think about how we could evaluate this recommender? In Chapter 9 we talked about doing cross-validation of the data but that doesn't really work for content data. So, to evaluate this recommender we can use the same code as used to evaluate the Mean Average Precision. Well almost.

We need to make a recommender method, which takes a list of ratings such that we can call it with the training part of the user's ratings. But we already had that, as shown in Listing 10.14. Now the next thing to consider is how you want to divide each user's data. Again, we can either do it such that we ensure that there is always a certain number of training data, or always a certain number of test ratings. Or it could be some specific point in time.

You can run the evaluation by executing the following:
Listing 10.15: executing the evaluation.

```
> python -m evaluator.evaluation_runner -cb
```

It will create a csv file with the data used to show the evaluations.

The following figure shows how MAP looks:

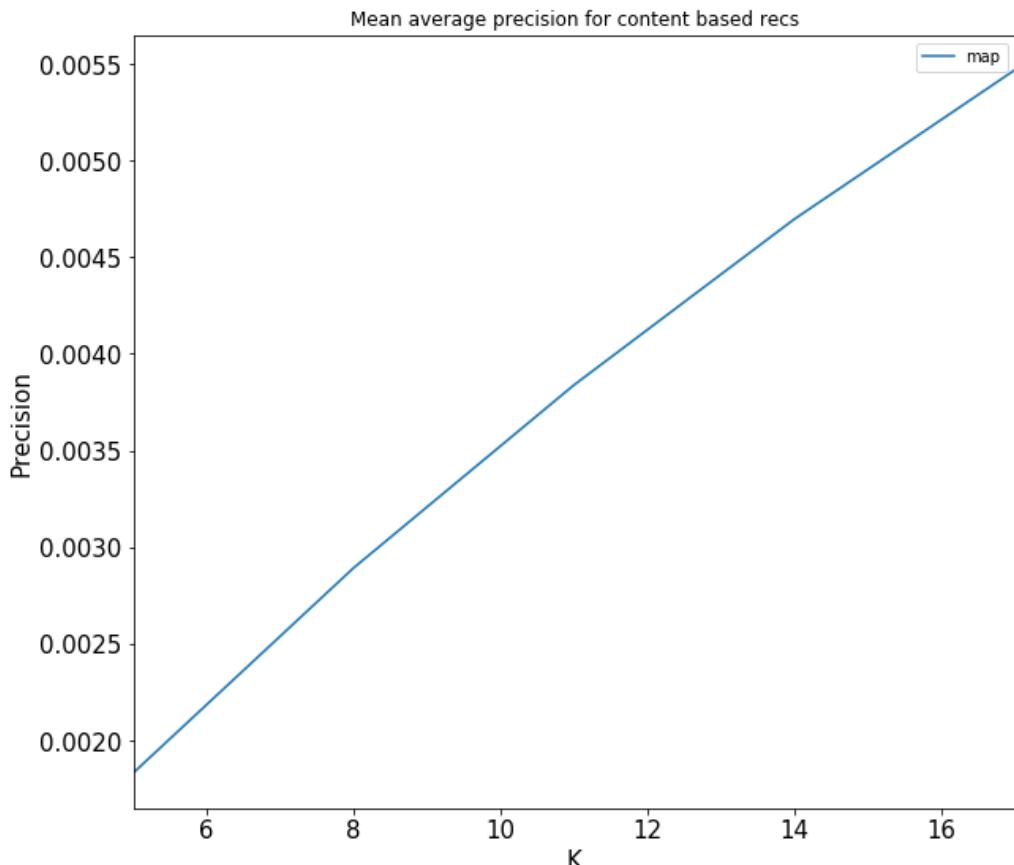


Figure 10.21 Evaluation on the content based recommender. Using these measures won't put the content based recommender in a very nice light.

Content-based recommenders works by finding similar content, the result of the evaluation will therefore depend on whether your domain and users are one where users typically stay within the same type of content. Or how adventurous the users are. Another way to test this type of recommender could also be by looking at how different recommended items are from the seed (input) content. The good thing is that all users that has rated one item, and there is an item similar to that will receive a rating. The test above is done using similarities at least above 0.1 and that produced recommendations to 96 % of the users. Which is a lot higher than a collaborative filtering algorithm.

10.12 Pros and Cons for content-based filtering.

Here are some things to consider when you build a content-based filtering algorithm

Pros

- **New items are easy to add**, just create the item feature vector and you are set to go.
- **You don't require a lot of traffic**. Since you can find similarity based on content descriptions, you can start recommending things from the first visit or rating.
- **It recommends across popularity** – content-based recommenders doesn't care which content is popular right now if it finds that a film nobody ever watched is just as likely to be recommended as one which everybody watches.

Cons:

- **New users are hard** – since we don't have any items to compare with
- **Conflates liking with importance** - This means that if you like science fictions films with Harrison Ford, the system will also give you films with Harrison Ford that are not.
- **No serendipity** – it is very specialized
- **Limited understanding of content** – It might be hard to include all features which mark the aspects that make content favorable to a user. Which means that the system can easily "misunderstand" what the user actually likes. In the example with the Thor movie, it could be that a user likes everything that comes out of the Shakespearian school, but normally dislikes action, and interpreting a user liking Thor because it is an action film. Or as Prof. Konstan says in his "introduction to Recommender Systems", if I like Sandra Bullock in action films and Meg Ryan in Comedies, But if I hate Meg Ryan in action films and Sandra Bullock in comedies, there is no way for that to be captured in the feature vector. That is unless you start combining them to have a feature "Action film starring Sandra Bullock" and "Comedy starring Sandra Bullock" etc.

10.13 Summary

We are doing great!

- TF-IDF is easy, I mean apart from remembering what it is acronym for⁷⁹. We can use it to find important words in documents.
- Before feeding descriptions and texts to an algorithm, it's good to remove unwanted words, and optimize for the algorithm. This can be done with removing stop words, too popular words, doing stemming, and also TF-IDF and removing words that are not important.
- Topic models create topics, which can be used to describe documents
- LDA makes the topic model.
- Evaluating content-based recommender can be done by dividing each user's ratings into train and test data as we learned in chapter 9. And then run through each user, calculate the recommendations and see if it produced something that was in the test set.
- Content-based recommender systems are good because they don't need a lot of knowledge about the user.
- Content-based recommender systems will find similar items, which might not always be the most surprising and fun recommendations to get.

If you are interested in content based recommenders, then I would suggest looking into also finding similar items using word vectors like the word2vec. I have seen some excellent results.

⁷⁹ Term Frequency – Inverse Document Frequency

11

Finding hidden genres with Matrix Factorization

The matrix is just numbers, this chapter is about the matrix and how to create them:

- You will go through dimensionality reduction recommender algorithms
- Reducing similarity will help you find latent (hidden) factors in the data
- You will train and use a Single Value Decomposition (SVD) to create recs.
- Having a SVD model, you will also learn how to fold in new users and items into an SVD
- You will also look at another matrix factorization model called the Funk SVD, which is more flexible than the original SVD.

11.1 Introduction

What have we learned so far?

In chapter 8, we looked at collaborative filtering using neighbor-based filtering. In this chapter, we are going to return to collaborative filtering, but this time we won't be talking about neighborhoods. Instead, we'll explore latent factors. In chapter 10 we also talked about latent factors, but at that point, we talked about latent factors in the content data. Now we will look at latent factors in relation to collaborative filtering.

Many names are being thrown around. But let's get it settled *hidden genres* is the same as *Latent factors*. At least when we talk about movies. The factors are said to be latent because it is not something that is defined by humans but something that an algorithm calculates, it is trends in the data that shows or explains the users' taste. These trends or factors are also

latent because, even if, they will make sense data-wise it might not be so easy to say that these factors mean. But we will try to explain stuff as we go along.

Latent factor recommenders are a relative new invention and got their real breakthrough when the Netflix prize competition promised 1 million dollars to anyone who could improve Netflix's recommendations by at least 10 percent. We will look at some of the solutions that were close to winning the prize. The winner was an ensemble recommender algorithm, which means mixing lots of different algorithms to produce the final result (and incidentally the topic of chapter 12). The winning ensemble was so complicated that it never actually came into production. Instead another solution made by a guy called Simon Funk has become quite famous for getting very close to winning and because he actually blogged about it. His solution has been the basis for many other solutions since.

In this chapter, we will concentrate on solutions based on the rating matrix. If you only have behavioral data or implicit ratings, you should first go through the steps of interpreting it into ratings, as we did in chapter 4. The FunkSVD⁸⁰, which we talk about in the end of the chapter, could be modified to use behavioral data instead of ratings.

Before we move on, I want to explain myself a bit. We will start out with a quite a lot of pages about a method called Single Value Decomposition (SVD). The SVD is a well-known method from linear algebra, and there are lots of tools out there to help you calculate one. I will show you with sklearn. With a true SVD you can add new users quite easily. However, it is terribly slow to calculate an SVD⁸¹, and if you have a large dataset, then it will be very hard to calculate the SVD. On top of that, there are some quite heavy requirements about what should be done about the empty cells in the rating matrix. Hence, we move on to another matrix factorization method called FunkSVD, which is becoming the more used one. It's not as easy to add new users, but it can be done.

Finding latent factors is a task that can be done in many ways; in the scope of collaborative filtering, they have been mostly done with matrix factorizations, based on the rating matrix.

11.2 Sometimes it's good to reduce the size of the data

Why it is good to reduce the data to lose the noise.

Everybody is always ranting about how it is good to have more data! What's up with this? Is this some sneaky attempt to go against the masses? Let me put you to ease and tell you up front that it isn't. But, we need to look at how to get most out of the data we do have. Firstly, a reason to reduce the dimensions could be to extract the signal from the data. For example, the top plot in figure 11.1 shows a scatter plot of some noisy data, while the bottom one

⁸⁰ It is a funky algorithm, but it has its name due to Simon Funk who popularized it.

⁸¹ <http://sysrun.haifa.il.ibm.com/hrl/bigml/files/Holmes.pdf>

shows the actual signal, i.e. the information in the data. Simplifying the data can sometimes make it easier to understand the information hidden in it.

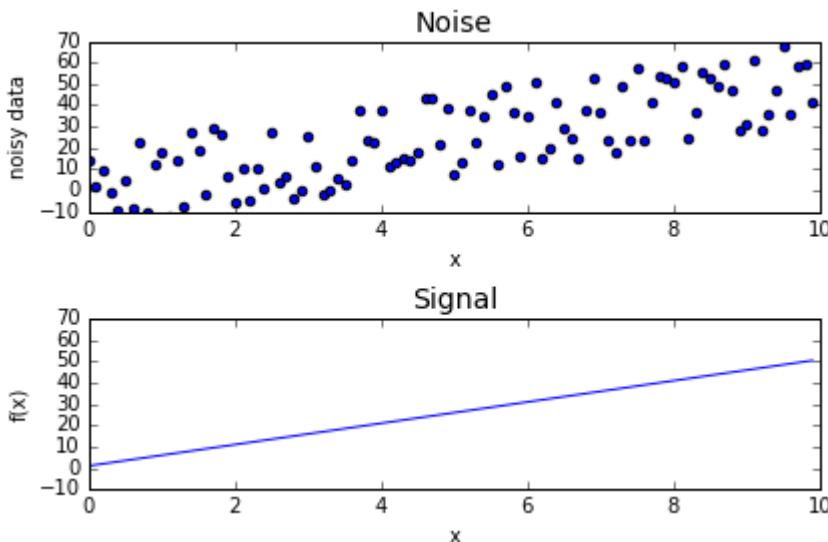


Figure 11.125 (Top) Noisy data, **(Bottom)** The signal in the data

In a sense, you could have the same information as points on a line. The same idea is used then you do dimension reduction, we have some data in a high dimension, data like that is often called a point cloud. A dimension reduction algorithm will then try to find the directions of the data that provide more information. If the algorithm succeeds, then the vectors pointing that direction is said to be a latent factor.

To illustrate this in an example more real to us recommender people, let's look at a story from the real world. Imagine that you are on a first date, or stuck in an elevator with somebody (stress levels being equally high). To release some of the tension, you start talking about movies. It might start with something like "Did you see X?" Or, it might be that you are in for a longer session and want to talk about movie tastes, and say "I like films with people dressed from a particular designer and with a slightly supernatural aspect, and preferable it makes me laugh and take place in the seventies." The other person says, "Oh, so you like *James Bond!*" "Yes, but also *A Single Man* [movie from 2009 directed by Tom Ford, who usually designs sunglasses and stuff] or *Clueless* [an even older comedy]." The other person says cool and goes off in some rant about *Dawson's Creek* (television series that stopped in 1998) being such a cool series because they all wore clothing of a very well-defined taste.

If we should do the same story with the usual set of genres, it would go something like this: "I like action movies, but also drama and comedy." And the other would say, "Yes, I like TV-series." This probably wouldn't turn out to be a very fruitful conversation.

The basic idea is that by looking at behavioral data of users, you can find categories or topics that will enable you to explain users taste with much less granularity than each movie, but rather something like factors that can position movies that users like close to each other.

Talking about hidden things, makes it sound like there is always something there, so it is, of course, worth to point out that if your data is random or it doesn't have any signal, the reduction won't provide any more information. On the other hand, by trying to extract factors from data you will also use more of the data collected for a user. The neighborhood algorithms discussed in chapter 8 would only use small parts of data when calculating predictions; here we will use more.

11.3 Example of what we want to solve

So, let's get back to the rating matrix that I have been dragging around for the last several chapters.

Table 11.16 Rating matrix



	Comedy	Action	Comedy	Action	Drama	Drama
Sara	5	3		2	2	2
Jesper	4	3	4		3	3
Therese	5	2	5	2	1	1
Helle	3	5	3		1	1
Pietro	3	3	3	2	4	5
Ekaterina	2	3	2	3	5	5

Again, we will do a dimension reduction by factorizing the rating matrix (which will be described below) to help find important factors so that we have the users and items in the same space and we can find films that are interesting to the user simply by finding which are near. We can also find similar users.

Using the matrix factorization, I could produce the plot shown in figure 11.2. I have used only two dimensions, first you can say that the vertical axis indicates how serious the films are, *Men In Black* (MiB) and *Ace Ventura* (AV) are not very serious, while *Les Miserables* is quite serious. It is important to note that this is just my interpretation of it. I did not try to fit the data to make it do a better factorization, so it's okay to be a bit puzzled. I did try to come

up with some interpretation of the horizontal axis, but having *Ace Ventura* and *Sense and Sensibility* at the same point made it very hard⁸².

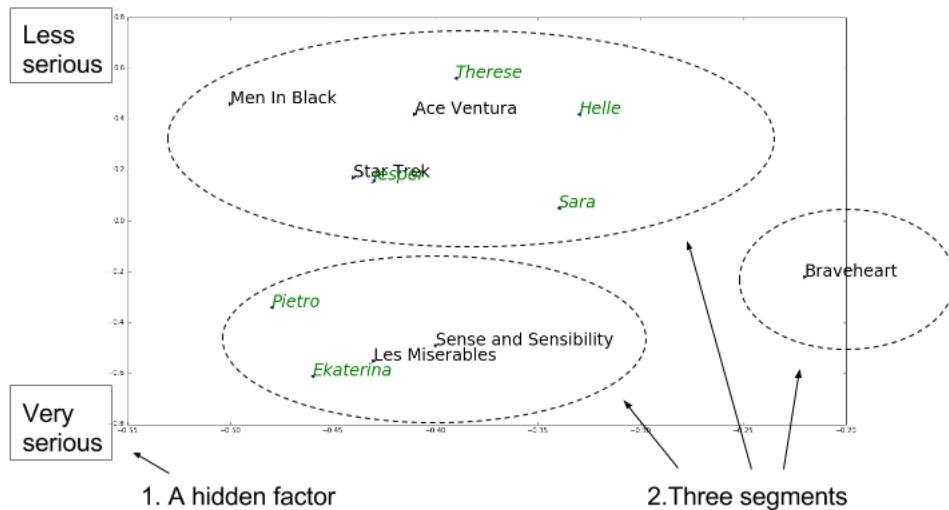


Figure 11.2 Plotting the movies and users in the reduced dimensional space. The reduced space has two dimensions. One seems to be the seriousness of the movies (y-axis), while I can't see what interpretation to put on the x-axis. It is important to note that the system didn't choose the number of dimensions, I did to make the result illustrable. More on that later. The visualization shows three segments, which seems to correspond well with the rating matrix.

- 1) The top one contains Therese, Helle, Jesper (hard to see as he is under the Star Trek) and Sara all likes comedies and action movies while 2) Pietro and Ekaterina are down with the dramas. 3) The third segment only contains Braveheart which is a bit outside of both. Looking at Figure 11.3 where the segments are pointed out in the rating matrix makes it easy to see why there would be these three segments.

⁸² Please let me know on the book forum if you come up with something.



Figure 11.326 Showing the three segments in the rating matrix, which is shown in figure 11.2

When I first saw this output, it puzzled me that Jesper's position was on top of Star Trek (below). He didn't rate the Star Trek movie that high, so why there. However, if you consider Jesper's ratings, he has given four starts to MiB and AV, while three starts to Star Trek and the two dramas. Then it seems sensible that Jesper's position is there, since it attempts to push items and users which are related closer. He is closest to MiB and AV, but still not too far from the two dramas.

Segment two also deserves another word. I think that Braveheart has won its own segment in the reduced space simply because it's the one film that all agree should be low. But again that is just my interpretation.

Calculating recommendations are done simply by looking into this factor space (the coordinate system shown in figure 2), and find the ones that are closer to the user. Before you start plotting your factor space and looking for clusters, then please listen to this advice: You will find a lot of tutorials and descriptions showing how you can interpret dimensions in a vector space into something understandable, but it is rarely something that really works, use a vector space as a box which you ask about similarity among items and users, that is what it is good at.

Even if they are hard to interpret then I'm really excited about this. It is so cool! Let's get started. I hope that this got you so interested that you are ready to learn a bit of math, because we need to dip a bit into Linear Algebra.

11.4 A whiff of linear algebra

Linear algebra is a large field of mathematics. It covers the study of lines, planes, and subspaces, but is also concerned with properties common to all vector spaces. Ideally the concept of matrix factorization is based on a lot more linear algebra than what we are going to look at here.

11.4.1 Matrix

which is Latin for "womb"⁸³

We have looked at vectors, which represent user taste, but they can also mean so many other things. If you have two vectors like

$$\boldsymbol{v}_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \text{ and } \boldsymbol{v}_2 = \begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix}$$

you could consider them as to arrows pointing in two directions, then you could draw a plane that passes through them, you can say they span a plane, like the plane shown in figure 11.3.

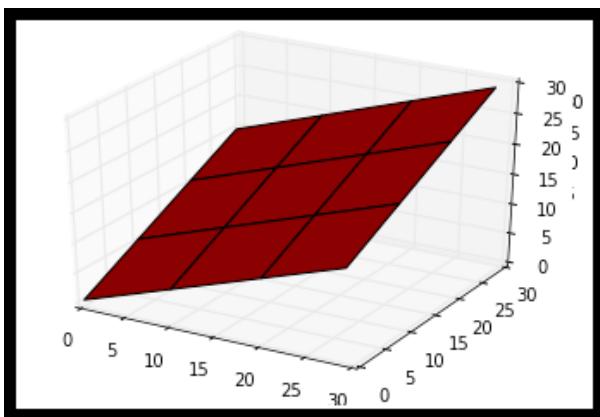


Figure 11.427 A plane spanned by two vectors.

⁸³ Read more interesting facts on linear algebra here: https://en.wikipedia.org/wiki/Linear_algebra

A way to represent two or more vectors could be as a matrix. The matrix M represents the vectors from above

$$\mathbf{M} = \begin{bmatrix} 1 & 0 \\ 1 & 2 \\ 1 & 1 \end{bmatrix}$$

A matrix is defined as a rectangular array of numbers, and declared by a number of rows m and a number of columns n. A vector is a special case of a matrix, which has only one column.

Now imagine our rating vectors for our users, we have a dimension for each content item, which means that we are talking about a vector with thousands of dimensions, and in a thousand-dimensioned space we will have something called a hyperplane, which all the vectors lie in. A matrix is a way of describing one of those planes or hyperplanes.

The rating matrix shown a couple of pages back will span some kind of hyperplane in a 6-dimensional space, and will probably be very hard to imagine, or draw. So, we won't. Just think of them as something in a space:

$$\mathbf{R} = \begin{bmatrix} 5 & 3 & 0 & 2 & 2 & 2 \\ 4 & 3 & 4 & 0 & 3 & 3 \\ 5 & 2 & 5 & 2 & 1 & 1 \\ 3 & 5 & 3 & 0 & 1 & 1 \\ 3 & 3 & 3 & 2 & 4 & 5 \\ 2 & 3 & 2 & 3 & 5 & 5 \end{bmatrix}$$

But before you get too confident with this way of seeing the rating matrix as a matrix, you should remember that we don't have values to put in all the cells, and you cannot have a matrix where only some of the cells are filled out. In the matrix above I have filled in zeros, but that is not the best thing to do. We will see later that quite a lot of thought has to go into deciding what should be put in those empty cells. In the following we will multiply matrices together. If you know how that words it great, but here is a quick example of how to do it.

MATRIX MULTIPLICATION

If you have two matrices U and V can be multiplied if A has the same number of columns as B has rows. Figure 11.5 shows an example about how to create a matrix by multiplying two together.

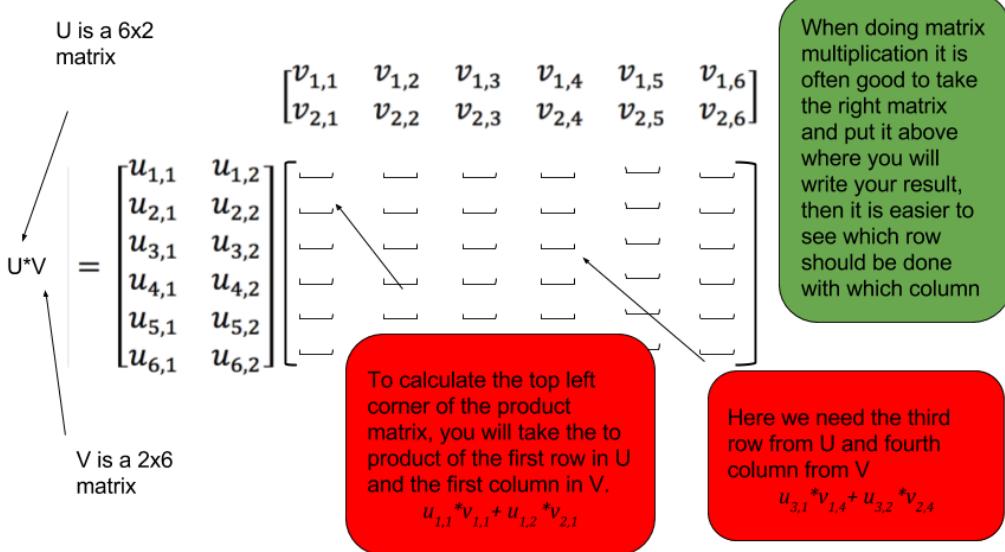


Figure 11.5 Crash course in matrix multiplication

The idea is that each cell in the new matrix is the dot product between the corresponding row in the first matrix (U in figure 11.5) and the corresponding column in the second matrix. With this quick example on to factorization.

11.4.2 What is factorization?

As discussed above, we want to factorize the matrix. Factorization is about splitting things up so for example you can split a number like 100 into the following prime factorization⁸⁴:

$$100 = 2*2*5*5.$$

It means that you take a number and write it as a number of factors, in this case as a list of prime numbers⁸⁵.

In our case, we don't have a number but a rating matrix. And what we can do in that case is that we can factor it into a product of matrices, so if we have a matrix R (R for rating matrix, blink blink) we should be able to decompose it into the following form

⁸⁴ <http://mathworld.wolfram.com/PrimeFactorization.html>

⁸⁵ Interestingly enough, then prime number factorization is unique, due to the fundamental theorem of arithmetic.

$$R=UV$$

If R has n rows and m columns (as in n users and m items), we call it an $n \times m$ matrix (read "n by m"), the size of U will be a $n \times d$ matrix and V is a $d \times m$ matrix. If we look at the matrix shown above we will have a formula as follows:

$$\begin{bmatrix} 5 & 3 & 0 & 2 & 2 & 2 \\ 4 & 3 & 4 & 0 & 3 & 3 \\ 5 & 2 & 5 & 2 & 1 & 1 \\ 3 & 5 & 3 & 0 & 1 & 1 \\ 3 & 3 & 3 & 2 & 4 & 5 \\ 2 & 3 & 2 & 3 & 5 & 5 \end{bmatrix} = \begin{bmatrix} u_{1,1} & u_{1,2} \\ u_{2,1} & u_{2,2} \\ u_{3,1} & u_{3,2} \\ u_{4,1} & u_{4,2} \\ u_{5,1} & u_{5,2} \\ u_{6,1} & u_{6,2} \end{bmatrix} \begin{bmatrix} v_{1,1} & v_{1,1} & v_{1,1} & v_{1,1} & v_{1,5} & v_{1,6} \\ v_{2,1} & v_{2,2} & v_{2,3} & v_{2,4} & v_{2,5} & v_{2,6} \end{bmatrix}$$

This is called a UV-decomposition. And here we set the d to be 2 it could also have been three or four or even five. But the idea is that we want to decompose the matrix R into some hidden features (read columns for users and rows for items) both for content items but also for the users.

In the field of recommender systems then we would call the U for the user-feature matrix and the V for the item-feature matrix.

Matrix factorization

To do the factorization we need somehow to insert values in the U and V matrix, in such a way that UV is as close to R as possible. Let's start simple, let's say we want to fit the cell in the second row, second column from the upper left corner of R as shown in figure 11.6.

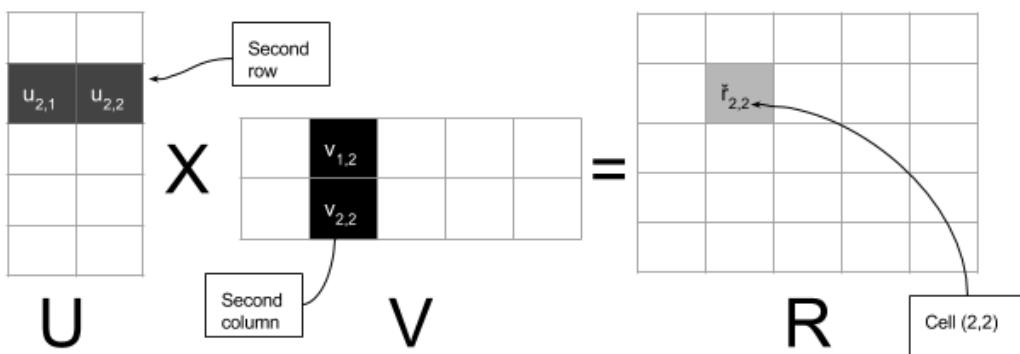


Figure 11.6 To multiply two matrixes we need to take the row and multiply it pairwise with the corresponding column and then add the products afterwards

As we saw above the cell is calculated using the dot product. The dot product is not new to the careful reader, we looked at it before, but let's refresh anyway. If you have the row and the column above, then you have two vectors $(u_{2,1}, u_{2,2})$ and $(v_{1,2}, v_{2,2})$, the dot product is simply

$$\hat{r}_{2,2} = u_{2,1} * v_{2,1} + u_{2,2} * v_{2,2}$$

If the vectors are longer, it works the same way. We will look at two ways to do the factorization. In the following we will look at the SVD method, which is an old mathematical construct which is well known and understood. After the SVD we will move onto something they call FunkSVD, which is a way to also create a factorization but in a different way. First the SVD

11.5 Constructing the factorization using SVD

One of the most predominantly used methods for matrix factorization is an algorithm called Singular Value Decomposition (SVD).

We want to find items to recommend to users, and we want to do it using extracted factors from the rating matrix. The idea of factorization is even more complicated because we want to end up with a formula that enables us to add new users and items without too much fuss.

From the rating matrix M , we want to construct two matrices that we can use, one which represents the customers taste and one that contains the item profiles.

Using Singular Value Decomposition (SVD), you construct three matrices: U , Σ , and V^T (aka V^* depending on which book you look in). We would like to end up with two, so you can multiply the square root of the Σ on the two others, and then you only have two left. But before doing that you want to use the middle matrix will give us information about how much we should reduce the dimensions. Figure 11.7 shows the SVD

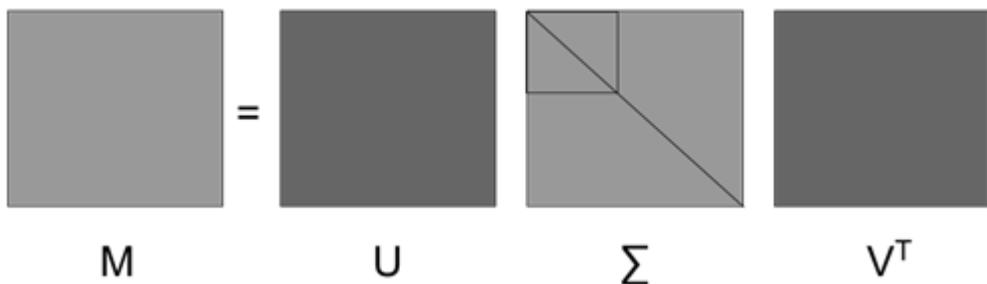


Figure 11.7 A matrix can be factorized into three matrices.

We call them:

M	A matrix we want to decompose, in our case it's the rating matrix.
U	user feature matrix
Σ	weights diagonal
V^T	item feature matrix

When using the SVD algorithm the Σ will always be a diagonal matrix.

DIAGONAL MATRIX

Diagonal matrix means that it only has zeros except for in the diagonal from upper left corner to lower right corner, like the following example:

$$\begin{bmatrix} 9 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

REDUCING THE MATRIX

It might be hard to see how splitting a matrix into three matrices can do any good, especially when I mention that it is very time consuming to create these. But the idea is that the central diagonal matrix contains elements in that are sorted from the largest to the smallest, the elements are called singular values and the values indicates how important this feature is for the data set. Where a feature here means both a column in the user matrix U and a row in the content matrix V^T . We can now select a number r of features and set the rest of the diagonal to zero, look at figure 11.8, which illustrates what will remain of the matrices when you set diagonal values to zero outside of the gray box. This is same as removing all right most columns in the user matrix U , and all the bottom rows from V^* keeping only the r top rows.

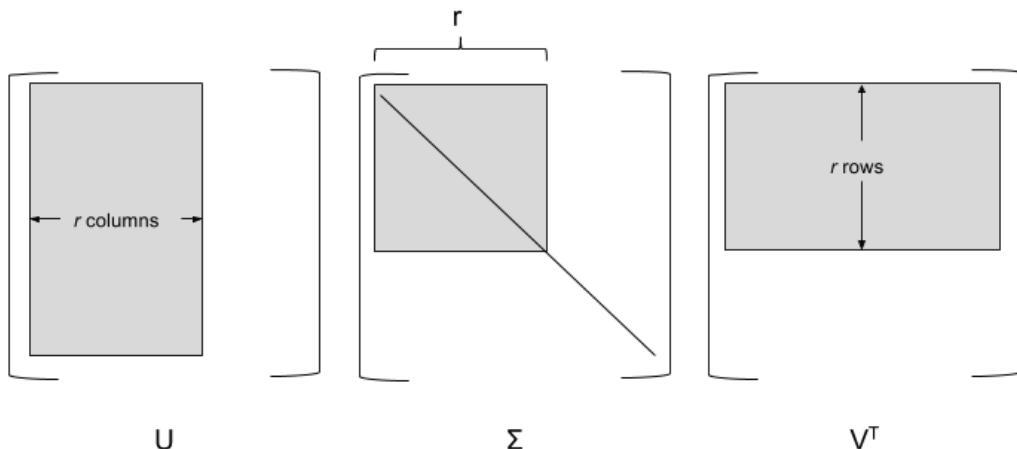


Figure 11.8 Reducing the SVD by setting small values in Σ to zero.

Let's look at a little example using the rating matrix as shown above. We can make a python panda Dataframe in the following way:

Listing 11.1: Creating a rating matrix

```
import pandas as pd
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/practical-recommender-systems>

Licensed to Asif Qamar <asif@asifqamar.com>

```

import numpy as np

movies = ['mib', 'st', 'av', 'b', 'ss', 'lm']
users = ['Sara', 'Jesper', 'Therese', 'Helle', 'Pietro', 'Ekaterina']

M = pd.DataFrame([
    [5.0, 3.0, 0.0, 2.0, 2.0, 2.0],
    [4.0, 3.0, 4.0, 0.0, 3.0, 3.0],
    [5.0, 2.0, 5.0, 2.0, 1.0, 1.0],
    [3.0, 5.0, 3.0, 0.0, 1.0, 1.0],
    [3.0, 3.0, 3.0, 2.0, 4.0, 5.0],
    [2.0, 3.0, 2.0, 3.0, 5.0, 5.0]],
    columns=movies,
    index=users)

```

Just for fun we can test out if it worked correctly and try out the following command

`M['mib']['sara']`, it prints 5.0, which is correct.

Listing 11.2: Doing a SVD on the matrix.

```

from numpy import linalg      ①
U, Sigma, Vt = linalg.svd(M) ②

```

① import the linear algebra library of numpy⁸⁶
 ② calculate the matrix factorization.

This will create the 3 matrices shown in figure 11.9 (which looks like what we showed in 11.6 and 11.7):

<table border="1"> <tbody> <tr><td>-0.34</td><td>0.05</td><td>0.91</td><td>0.11</td><td>0.19</td><td>-0.00</td></tr> <tr><td>-0.43</td><td>0.16</td><td>-0.31</td><td>-0.12</td><td>0.74</td><td>0.35</td></tr> <tr><td>-0.39</td><td>0.56</td><td>-0.19</td><td>0.63</td><td>-0.32</td><td>0.02</td></tr> <tr><td>-0.33</td><td>0.42</td><td>0.02</td><td>-0.76</td><td>-0.37</td><td>-0.05</td></tr> <tr><td>-0.48</td><td>-0.34</td><td>-0.18</td><td>0.03</td><td>0.10</td><td>-0.78</td></tr> <tr><td>-0.46</td><td>-0.61</td><td>-0.06</td><td>0.02</td><td>-0.40</td><td>0.51</td></tr> </tbody> </table>	-0.34	0.05	0.91	0.11	0.19	-0.00	-0.43	0.16	-0.31	-0.12	0.74	0.35	-0.39	0.56	-0.19	0.63	-0.32	0.02	-0.33	0.42	0.02	-0.76	-0.37	-0.05	-0.48	-0.34	-0.18	0.03	0.10	-0.78	-0.46	-0.61	-0.06	0.02	-0.40	0.51	<table border="1"> <tbody> <tr><td>17.27</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>5.84</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>3.56</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>3.13</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1.67</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0.56</td></tr> </tbody> </table>	17.27	0	0	0	0	0	0	5.84	0	0	0	0	0	0	3.56	0	0	0	0	0	0	3.13	0	0	0	0	0	0	1.67	0	0	0	0	0	0	0.56	<table border="1"> <tbody> <tr><td>-0.50</td><td>-0.44</td><td>-0.41</td><td>-0.22</td><td>-0.40</td><td>-0.43</td></tr> <tr><td>0.46</td><td>0.17</td><td>0.42</td><td>-0.22</td><td>-0.49</td><td>-0.55</td></tr> <tr><td>0.50</td><td>0.22</td><td>-0.78</td><td>0.26</td><td>-0.08</td><td>-0.13</td></tr> <tr><td>0.34</td><td>-0.77</td><td>0.17</td><td>0.51</td><td>-0.02</td><td>-0.01</td></tr> <tr><td>0.41</td><td>-0.36</td><td>-0.16</td><td>-0.76</td><td>0.19</td><td>0.25</td></tr> <tr><td>-0.01</td><td>-0.03</td><td>0.01</td><td>-0.02</td><td>0.75</td><td>-0.66</td></tr> </tbody> </table>	-0.50	-0.44	-0.41	-0.22	-0.40	-0.43	0.46	0.17	0.42	-0.22	-0.49	-0.55	0.50	0.22	-0.78	0.26	-0.08	-0.13	0.34	-0.77	0.17	0.51	-0.02	-0.01	0.41	-0.36	-0.16	-0.76	0.19	0.25	-0.01	-0.03	0.01	-0.02	0.75	-0.66
-0.34	0.05	0.91	0.11	0.19	-0.00																																																																																																									
-0.43	0.16	-0.31	-0.12	0.74	0.35																																																																																																									
-0.39	0.56	-0.19	0.63	-0.32	0.02																																																																																																									
-0.33	0.42	0.02	-0.76	-0.37	-0.05																																																																																																									
-0.48	-0.34	-0.18	0.03	0.10	-0.78																																																																																																									
-0.46	-0.61	-0.06	0.02	-0.40	0.51																																																																																																									
17.27	0	0	0	0	0																																																																																																									
0	5.84	0	0	0	0																																																																																																									
0	0	3.56	0	0	0																																																																																																									
0	0	0	3.13	0	0																																																																																																									
0	0	0	0	1.67	0																																																																																																									
0	0	0	0	0	0.56																																																																																																									
-0.50	-0.44	-0.41	-0.22	-0.40	-0.43																																																																																																									
0.46	0.17	0.42	-0.22	-0.49	-0.55																																																																																																									
0.50	0.22	-0.78	0.26	-0.08	-0.13																																																																																																									
0.34	-0.77	0.17	0.51	-0.02	-0.01																																																																																																									
0.41	-0.36	-0.16	-0.76	0.19	0.25																																																																																																									
-0.01	-0.03	0.01	-0.02	0.75	-0.66																																																																																																									
U	Σ	Vt																																																																																																												

Figure 11.9: The matrices of the factorization

Does that make sense? Well not really, probably? It's so much work to get 3 matrices with exactly the same size as the original. But, hold your horses a second. Look at the diagonal matrix in the middle (Σ), more significantly known as the weights matrix.

⁸⁶ <https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.svd.html>

Each of these weights can be an indication of how much information present in each dimension. So, the first one provides a lot of information (17.27), the next one 5.84, and so on, so we can reduce the size of the matrices. But how much?

HOW MUCH SHOULD IT BE REDUCED

We could reduce the dimensions and only use two, and be able to produce a chart like the one of figure 11.2. Another good reason for reducing it to 2 dimensions, is that looking at the weights in the Sigma matrix shows that we will get a lot of the information by just using two features.

With such a small example, it doesn't matter much. But a rule of thumb is that you should retain 90% of the information. So, if we add up all of the weights, that is a 100%, then you should continue counting weights until you have 90% of the information. Let's do the calculation for the matrix above.

The sum of all of them is 32.0, so 90% of that is 28.83. If you reduce it to 4 dimensions, then the weight is 29.80 so we should reduce it to 4.

To reduce the matrices in code you'd do the following:

Listing 11.3: Reducing the matrix.

```
def rank_k(k):
    U_reduced= np.mat(U[:, :k])
    Vt_reduced = np.mat(Vt[:, :k])
    Sigma_reduced = Sigma_reduced = np.eye(k)*Sigma[:k]
    return U_reduced, Sigma_reduced, Vt_reduced,
```

①

```
U_reduced, Sigma_reduced, Vt_reduced = rank_k(4)
```

②

```
M_hat = U_reduced * Sigma_reduced * Vt_reduced
```

③

① Method who will return the reduced matrices.

② using the rank_k method return the reduced matrices

③ Calculate the deduced matrix M hat

The M_hat will look like the matrix shown in figure 11.10.

	mib	st	av	b	ss	lm
Sara	4.87	3.11	0.05	2.24	1.94	1.92
Jesper	3.49	3.46	4.19	0.95	2.62	2.82
Therese	5.22	1.80	4.92	1.59	1.10	1.14
Helle	3.25	4.77	2.90	-0.47	1.14	1.13
Pietro	2.93	3.05	3.03	2.11	4.30	4.67
Ekaterina	2.27	2.77	1.89	2.50	4.92	5.35

Figure 11.10 Showing the M_hat matrix

It is probably tempting to throw away the U and V matrices, but if a new user arrives we can add them to the model, but only if we have the factorized matrices. Let's see how to predict a rating.

PREDICT A RATING

With the factorization in place, it is now easy to predict ratings for users. Simply look it up in the new M_hat matrix. To look up something in a matrix you first denote the column (in the following example we look at 'av') and then index of the row (here 'sara'). The following shows a look up (okay after it wraps the M_hat in a DataFrame with nice column and index names):

Listing 11.4: Predicting a rating:

```
M_hat_matrix = pd.DataFrame(M_hat, columns= movies, index= users ).round(2) ①
M_hat['av']['sara'] ②
```

- ① Wrap the M_hat matrix in a dataframe so we can query it as we did with the rating matrix.
- ② Querying for sara's predicted rating for Ace Ventura.

The M_hat matrix contains all the predicted ratings so to calculate a rating prediction is a matter of looking up in the M_hat matrix. But we wanted to save the decomposed matrices only. To avoid to do three multiplications, you will take the Σ matrix and find the square root of it and then multiply it to each of the matrices, you update the matrix reduction matrix above to the following:

Listing 11.5: Reducing the matrix.

```
def rank_k2(k):
    U_reduced= np.mat(U[:, :k])
    Vt_reduced = np.mat(Vt[:, :k, :])
    Sigma_reduced = Sigma_reduced = np.eye(k)*Sigma[:, :k]
    Sigma_sqrt = np.sqrt(Sigma_reduced) ①
    ②
```

```

    return U_reduced*Sigma_sqrt, Sigma_sqrt*Vt_reduced
U_reduced, Vt_reduced = rank_k2(4)           ③
M_hat = U_reduced * Vt_reduced               ④

```

- ① method to reduce the size of the decomposed matrixes
- ② taking the square root of all entries.
- ③ call the method to get the reduced matrices.
- ④ To produce the $M_{\hat{}}^{\text{hat}}$ matrix simply multiply the two.

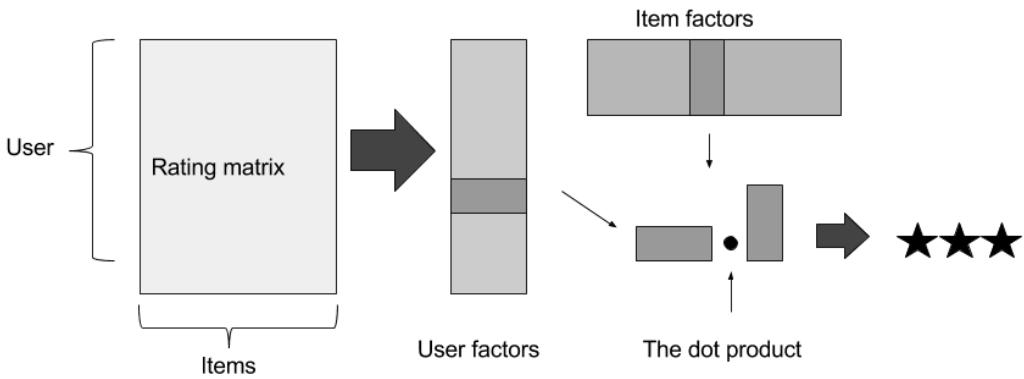


Figure 11.11 how to predict ratings using the factors

Now with these two we predict a rating like the following (and illustrated in figure 11.11). Here we are indexing the matrix in the shape of Numpy arrays and not as DataFrames as we did above.

Listing 11.6: Calculating ratings

```

Jesper = 1                                     ①
AceVentura = 2                                 ①
U_reduced[Jesper]*Vt_reduced[:,AceVentura]     ②

```

- ① Setting variables to make it more readable
- ② Calculating a predicted rating.

Running the code above shows that Jesper would rate Ace Ventura to 4.19, which is very close to the actual rating. But if we instead try to produce a rating for Sara on the same film we get 0.048, which is also very close to the zero we feed into the algorithm. So, maybe filling all the empty spots with zeros wasn't such a good idea. To make the predictions better we will do something called imputation.

HOW TO SOLVE THE PROBLEM OF THE ZEROS IN THE RATING MATRIX USING IMPUTATION:

What do we do with all the data we don't know? The example we looked at above only has few unknowns, but often we talk about situations where only 1% of the cells in the rating matrix have values. Something needs to be done. There are two common ways to approach this;

- You calculate the mean of each item (or user⁸⁷) and fill out this mean where there are zeros in each row (or column).
- Normalize each row, such that all elements will be centered around zero, so the zeros will become the average.

You can either inject average values into all the empty fields or use a mean.

Both methods are known as Imputation. This solution will take you some of the ways, but we can do better with something called baseline predictors, which we will talk about in a few sections time. Here we will just fill out the cells with zeros with the averages of the product ratings.:

Listing 11.7: Normalizing ratings

```
r_average = M[M > 0.0].mean()  
M [M == 0] = np.NaN  
M.fillna(r_average, inplace=True)
```

1
2
3

If we run it through the mill again, we get that Sara's predicted rating for *Ace Ventura* is 3.47. Which I think sounds closer to what it should be. It also sounds very close to the average rating for *Ave Ventura* which is 3.4.

11.5.1 Adding a new user by folding in

One cool thing about the SVD method is that we can also add new users and items into the system (though not before there have been some interactions). For example, we could add me to the factorization; my rating vector would look something like:

⁸⁷ but according to Sarwar et al. Application of Dimensionality Reduction in Recommender System -- A Case Study using the average of the item is best.

Table 11.2 ratings from Kim

	PROTECTING THE EARTH FROM THE SCUM OF THE UNIVERSE MEN IN BLACK	STAR TREK	ACE VENTURA: ANIMAL AGENT OF JUSTICE	BRAVEHEART	SENSE & SENSIBILITY	LES MISÉRABLES
	Comedy	Action	Comedy	Action	Drama	Drama.
Kim	4	5		3	3	

Or simply

$$\mathbf{r}_{\text{kim}} = (4.0, 5.0, 0.0, 3.0, 3.0, 0.0)$$

What we want to project this rating vector into our vector space, to do that we can utilize the decomposition above. Basically, to add a new user, means to add another row to the rating matrix, which in turn would mean that the decomposed user matrix would also have another row, as illustrated figure 11.12.

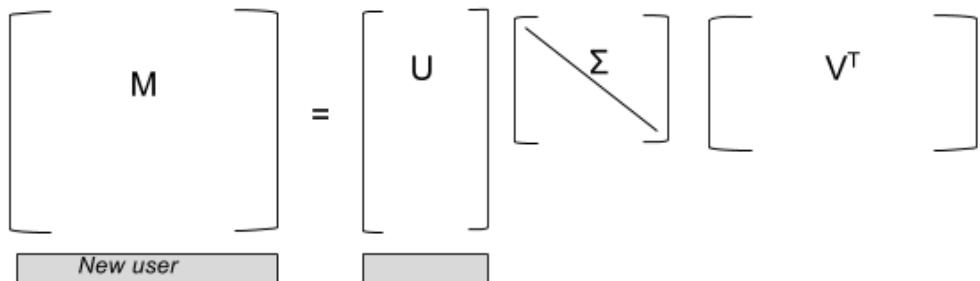


Figure 11.12 The schematics of the SVD folding-in technique

So how do we do that? Well simple, we just have to use the things we know already. We know the row of ratings, we know Σ and V^* . There are some rules about how to work with matrices that comes into play, but I will leave that out here since it will require a lot of writing about matrices⁸⁸. And just hope you will trust me that you can calculate the new row using the following equation:

⁸⁸ Have a look at Using Linear Algebra for Intelligent Information Retrieval for more details <http://lsirwww.epfl.ch/courses/dis/2003ws/papers/ut-cs-94-270.pdf>

$$u_{kim} = r_k V^t \Sigma^{-1}$$

u_{kim} - User vector in the reduced space representing the new user

r_k - new users rating vector.

Σ^{-1} - Inverse of the sigma matrix.

V^t - The item matrix.

Listing 11.8: Folding in new users.

```
from numpy.linalg import inv
r_kim = np.array([4.0, 5.0, 0.0, 3.0, 3.0, 0.0])
u_kim = r_kim * Vt_reduced.T * inv(Sigma_reduced)
```

now we can predict ratings for user Kim as well.

So if you can add new users and items, then why ever recalculate everything, you might ask. But remember that the reduction is done to extract topics from the data, these topics are not updated when you fold a new user in or item, they are just placed compared to the topics that were already there.

It is, therefore, important to update the SVD as often as possible, depending on how many new users and items you have, you should do it ones a day, or ones a week.

One curious thing of folding in a new user, is that if the new user only has one rating, then it doesn't matter what if it is high or low⁸⁹. The list of recommendations will be exactly the same. Try to play around with the small matrix above to realize why that is so.

11.5.2 How to do recommendations with SVD

Two ways we could make recommendations now, we could either say we calculate all predicted ratings and take the largest ones, that the user hasn't seen before. Or we can iterate through each item and find similar products in the reduced space. Or you can use the new matrices you have to calculate neighborhood collaborative filtering as we did in chapter 8. The reason why that might be a good idea is that the matrices have all non-zero entries (or at least if it is normalized), in this dense space you will have a much better chance of finding similar items or users.

⁸⁹ as pointed out in the interesting paper *Fifty Shades of Ratings: How to Benefit from a Negative Feedback in Top-N Recommendations Tasks* by E. Frolov et. Al.

PROBLEMS WITH SVD

I seem to be able to keep on writing about SVD and its possibilities but I want to move on to another type of reduction methods, which is similar to SVD, but much more efficient to calculate.

The SVD we have seen so far have some problems, it requires that something is done with the unfilled cells in the rating matrix. And with large matrixes it becomes very slow to calculate. On the good side there is the possibility to fold in new users as they arrive, but the SVD model is static and needs to be updated as often as possible.

SVD is not at all explainable. People like to know why something is recommended; the SVD approach makes it very hard for the machine to explain why it is predicting high ratings for one item and not another.

But before we move on I think you should have a look at the following article written by Sarwar, Karypis, Konstan, Riedl which is a group of people you should know if you are into recommender systems. The article is called *Application of Dimensionality Reduction in Recommender System -- A Case Study*⁹⁰

Before moving on to the next matrix factorization algorithm, let's sidestep a second and look at something called Baseline Predictors, these are used to make it easier for us to add values into the empty slots of the matrix. This can actually be used as a recommender system in itself, but here it is considered to be a way to make the matrix factorization better.

11.5.3 Baseline Predictors

Besides item types and the users' tastes, there are some other aspects of items and users we can talk about. If a movie is generally considered good, then the average rating of that film is probably slightly higher than the global average (the average of all movies) and in turn, if a film was considered bad its average rating would likely be below the global average. If we had information like that, then we could add a slightly higher default rating on the item. At the same time, some users are more critical than others (did I say grumpy old farts?), or more positive. An item that is above or below the average you say that it is biased, and the same with users, you can say that users have a bias compared to the global average.

If we could extract the biases for the items and the users, then we would be in a position to provide a baseline for our predictions, which is much better than just the average as we did above when filling out the rating matrix. Using these biases, we can create baseline predictors, a baseline predictor is the sum of the global average plus the bias of the item plus the bias of the user. In math you would write the following equation:

⁹⁰ <http://files.grouplens.org/papers/webKDD00.pdf>

$$b_{ui} = \mu + b_u + b_i$$

where

b_{ui} = base prediction of item i for user u

b_u = user bias

b_i = item bias

μ = the average of all ratings.

But that all sounds clever, but how do you calculate the user and item bias, because they are both part of the ratings. We have an equation for each rating. If for example we have that Sara rated the Avenger film "Civil War" only 3 out of 5 stars since she thought Capitan America is not acting nice, we will have the following equation:

$$b_{(sara, civil\ war)} = \mu + b_{sara} + b_{civil\ war} \Rightarrow 3 = 3.6 + b_{sara} + b_{civil\ war}$$

The global average of the example is 2.99 (sum all the cells, and divide with the number of nonzero cells). Can you answer what the values of the biases are? It's not possible, to say which is what, but if we have many equations with the biases, we can get close by trying to solve it as a least squares problem.

FINDING BIAS BY FINDING LEAST SQUARES

Back in chapter 7 on similarity, we talked about least squared when talking about similarity. It's the same idea here; we want to find biases which make the baseline predictions as similar to the ratings we know.

If we take the same rating as above we will ask what values should we set for the biases to make the following as small as possible:

$$\begin{aligned} \min_b (r_{(sara, civil\ war)} - b_{(sara, civil\ war)})^2 \Rightarrow \\ \min_b (r_{(sara, civil\ war)} - \mu - b_{sara} - b_{civil\ war})^2 \end{aligned}$$

To be sure I am not losing anybody here, I will just go quickly through this. The equation means that we are trying to find the b 's that makes the equation as small as possible, or simply the minimum. When we have many ratings (or at least more than one) we will find the minimum of the sum of all of them. The reason for the equation to be squared is that it will make them all positive. When we have a lot of ratings, we write it like the following:

$$\min_b \sum_{(u,i) \in K} (r_{(u,i)} - \mu - b_u - b_i)^2$$

Where

$(i,u) \in K$ means all ratings we have so far.

means all ratings we have so far.

A SIMPLER WAY TO CALCULATE BIASES

A simpler way to find these biases is to use the following equations:

First, calculate the bias for each user, taking the sum of the difference between the users rating and the mean. Divide this by the number of ratings, which means the result is the average difference between the mean and the users' ratings:

$$b_u = \frac{1}{|I_u|} \sum_{i \in I_u} (r_{u,i} - \mu)$$

When all the users' bias have been calculated, calculate the item bias the same way:

$$b_i = \frac{1}{|U_i|} \sum_{u \in U_i} (r_{u,i} - b_u - \mu)$$

These baseline predictors, as mentioned above, can be used to fill out the empty spaces of the rating matrix to make the SVD or indeed most matrix factorization algorithms work better. I have calculated the biases for the test data above and they are:

Listing 11.9: Calculating biases

```
global_mean = M[M>0].mean().mean()          1
M_minus_mean = M[M>0]-global_mean           2
user_bias = M_minus_mean.T.mean()             3
item_bias = M_minus_mean.apply(lambda r: r-user_bias).mean() 4
```

- ① finding the global mean, can be done by first finding the mean of each column and then find the mean of that.
- ② now deduct the global mean from all the nonzero ratings.
- ③ The mean of each row is the user bias
- ④ deducting the user bias from each row, and then taking the mean of each column and you have the item bias.

User bias:

Sara	-0.197222
Jesper	0.402778
Therese	-0.330556
Helle	-0.397222
Pietro	0.336111
Ekaterina	0.336111

Item bias

mib	0.644444
st	0.144444
av	0.333333
b	-0.783333
ss	-0.355556
lm	-0.188889

Now it is time for another way to do the factorization, which intelligently only focuses on the values we know.

11.5.4 Temporal dynamic

So far, we have talked about bias as something that is static, but a user could move from a happy rater to a grumpy old man, and then the biases should be adjusted to that, the same with the item bias should also be adjusted over time, as items go in and out of fashion. This also means that predictions of the ratings could vary over time, so you could say our rating prediction function is a function of time also. It is something to consider, if you implemented everything else, and want to try and squeeze more precision out of the recommender.

11.6 Constructing the factorization using FunkSVD

The SVD method above puts a lot of weight into the rating matrix, but the rating matrix is a sparse matrix and should in general not be trusted too much. In the sense that the chance of finding a cell that is populated with a rating can be below 1%. So instead of using the whole matrix Simon Funk came up with a method that only utilizes the things we know. We need to learn some more math and stuff to really learn to appreciate it. But it will be full of figures so that it will be over in no time. The FunkSVD is also often referred to as regularized SVD.

We will start out looking at a way that is called Root mean squared error (RMSE among friends), which we also talked about in chapter 7, to provide us a measure of how close we are to the known ratings. With that in our toolbox, we will look at something called Gradient Descent, which uses RMSE to walk (don't believe me, then better read on) towards a better solution. When we have that then we will look at how we can use baseline predictors here also, I already mentioned them, as a way to predict better than average rating information. And then when we have learned all that, then we will look at the actual FunkSVD algorithm.

11.6.1 Root Mean Squared Error

When we are talking about optimizing algorithms, our first answer should always be RMSE, which is the root mean squared error. Let's go through how that will make us happier. First, we compare the matrix that we get by multiplying U and V to see how different the values are.

$$\begin{bmatrix} 5 & 3 & 0 & 2 & 2 & 2 \\ 4 & 3 & 4 & 0 & 3 & 3 \\ 5 & 2 & 5 & 2 & 1 & 1 \\ 3 & 5 & 3 & 0 & 1 & 1 \\ 3 & 3 & 3 & 2 & 4 & 5 \\ 2 & 3 & 2 & 3 & 5 & 5 \end{bmatrix} = \begin{bmatrix} u_{1,1} & u_{1,2} \\ u_{2,1} & u_{2,2} \\ u_{3,1} & u_{3,2} \\ u_{4,1} & u_{4,2} \\ u_{5,1} & u_{5,2} \\ u_{6,1} & u_{6,2} \end{bmatrix} \begin{bmatrix} v_{1,1} & v_{1,1} & v_{1,1} & v_{1,1} & v_{1,5} & v_{1,6} \\ v_{2,1} & v_{2,2} & v_{2,3} & v_{2,4} & v_{2,5} & v_{2,6} \end{bmatrix}$$

So we have a long list of equations which we will look at. And for each of them we want the following to be as small as possible:

$$\min_{u,v} (r_{ui} - u_i \cdot v_j)$$

Since we want to do that for all cells we can make a sum of these expressions and say we want the minimum of all of them

$$\min_{u,v} \sum_{(u,i) \in \text{known}} (r_{ui} - u_u v_i)$$

Remember that each row was a user (u), and each column was a content item (i), and each cell in the matrix is defined by (u,i) . The user-item notation makes it more clear that is simply all the ratings that the user u has given, in other words, it's all the ratings we know already. The goal here is to find values for the two matrices that minimize the difference between actual rating and the one you'd calculate from U and V . A way to minimize the difference is to use a method called Gradient Descent.

We want the equation to penalize big errors so we take the square of the difference. But we still want the error to come out to be on the same scale of the ratings so with that we end up with the RMSE:

$$RMSE = \sqrt{\frac{1}{|\text{known}|} \sum_{(u,i) \in \text{known}} (r_{ui} - u_u v_i)^2}$$

We square each element of the sum, when everything is summed we divide with the number of elements in the sum, and then take the square root. We will use this in a little bit.

To make the explanation in the following more relevant, we will just point out that the RMSE above can be seen as function of the shape:

$$f(u_1, \dots, u_N, v_1, \dots, v_M) = \sqrt{\frac{1}{|\text{known}|} \sum_{(u,i) \in \text{known}} (r_{ui} - u_u v_i)^2}$$

And the goal is to find values $u_1, \dots, u_N, v_1, \dots, v_M$, which makes result of the function as small as possible. Since that is rather difficult to do just by looking at it, we will turn to something called Gradient Descent

11.6.2 Gradient Descent

To understand gradient descent let's start out with a general example, and then go back to the problem at hand.

Gradient descent is a way to find optimal points on a graph, where optimal means the lowest point (or the highest point). Consider we had the following function, which we wanted to find the x and y that produced the smallest possible value:

$$f(x,y) = 12x^2 - 5x + 10y^2 + 10$$

If we plot it, it looks like the one shown in figure 11.13.

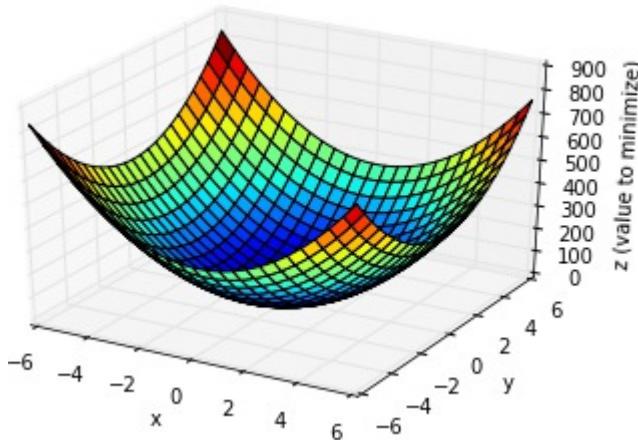


Figure 11.13: Plotting the function $f(x,y)=12x^2 - 5x + 10y^2+10$

The thinking behind gradient descent is to start somewhere (we will get back to how to select that somewhere) and then look around to see if there is any direction that makes the function produce a smaller value. In math language we want to find x' and y' such that:

$$f(x',y') < f(x,y)$$

In our example (seen in figure 11.12) it boils down to understanding what side of the “bowl” you are standing on and then moving in the direction that points towards the bottom. Consider yourself standing on a mountain, it's foggy, and you can only see a meter in each direction. If you should get down towards where to find water, then the best choice is probably to go in the direction that leads downwards. Same principle here.

In the following, we will look at how to translate that.

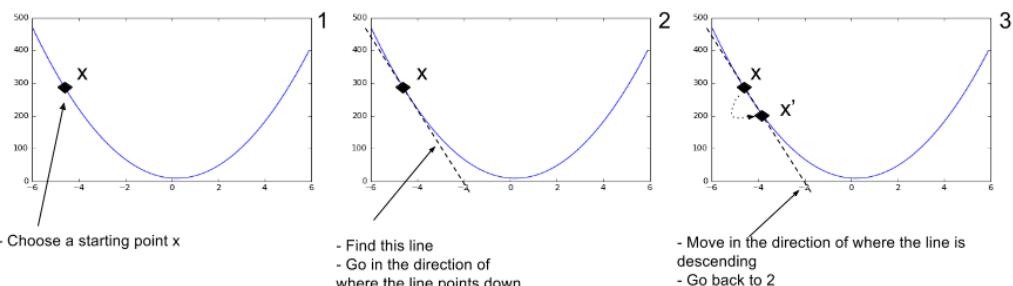


Figure 11.14 Gradient descent algorithm. Start out somewhere find the direction that moves down, move a little bit in that direction and repeat.

I will divide the description into some steps.

HOW TO START

How do you decide which point to start out with?

We can start anywhere because it is not all functions that are nicely bowl shaped like the one above. Often you will be looking at functions that have more than one local minimum like the one shown in figure 11.15.

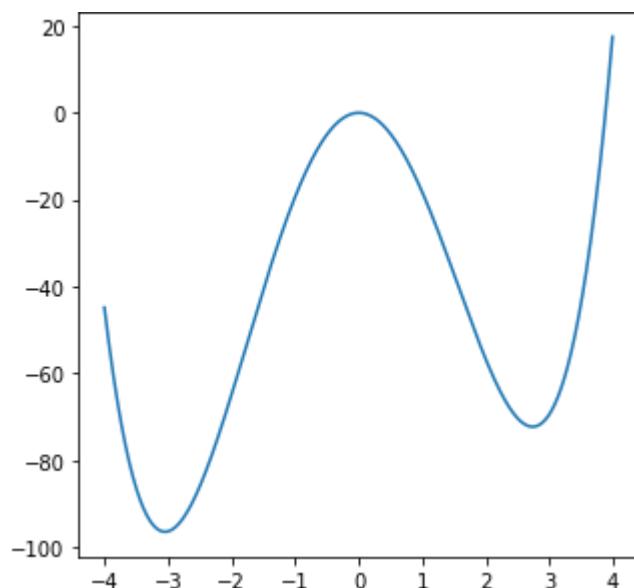


Figure 11.15 Function with more than one minimum ($1.2x^4+0.5x^3-20x^2$)

Often the best suggestion is to try out different starting points, and see if they arrive at the same point.

If we continue on the function above, and the steps in figure 11.14. we picked $x = -5$.

HOW TO FIND THE LINE THAT POINTS DOWN

To find the line pointing down we need to find the derivative. If you don't know about derivation then you will have to take my word on the fact that the derivate of $12x^2 - 5x + 10$ (this is the function after we set $y=0$) is the following:

$$dy/dx = 24x - 5$$

This means that if we input our chosen starting point ($x = -5$), we get $dy/dx=-125$ this means the original function at $x = -5$ will slope downward with a slope equal to a fall of 125 every time you move 1 to the right. So, if we want to find the x that gives the minimum result, we should go towards the right.

If the derivate function would produce something positive, we should have moved to the right to approach a minimum. To check what I am saying is true, we can look at figure 11.14, where we can see that if $x = 2$ then we should be going towards the left. If we insert $x = 2$ into the derivate, we get $dy/dx=43$ which mean we should go left.

If we are looking at a function with several variables like the one in figure 11.13, you do this trick for each of the variables.

HOW TO FIND THE NEXT POINT

So we know which way to go, now how do we move on to the next point. So how do we take the step?

$$x' = x - \alpha * \frac{dy}{dx}$$

Now don't stress just because I added a Greek letter, its just an alpha, and in the world of gradient descent it is called the *learning rate*, which translates to how big a step we should take every time.

Again there is not really any rules, other than if you take too large steps then you might miss the minimum all together. Looking at figure 11.14 step 3 if you step more than five you will miss the bottom and go back and forth between the sloping sides. If you do too small steps, you will never arrive.

If you have more than one variable, you do the same for each variable.

WHEN IS IT FINISHED

You can't say for sure that when you are finished either (this is why they call it a heuristics and not a solution). But what you can look for is when your scoring function - the function we try to optimize changes less and less. So if you have done a step, and the function only got 0.0001 better, then it might be time to stop. Alternatively, you can say that you continue for 150 iterations (steps) and when you are done, you are done.

11.6.3 Stochastic Gradient Descent

The gradient descent algorithm described above is also known as the Batch Gradient Descent because you calculate all the errors every time you move the value of the parameters, which can be quite a lot of work. Consider that in the dataset we are using contains no less 601263

ratings, hence every time the gradient descent algorithm does an iteration you needed to calculate all 601,263 subtractions.

Listing 11.10: Get count of all ratings using the Django models interface

```
In[1]: Rating.objects.all().count()
Out[1]: 601263
```

Another way, which has also proven both efficient and effective in speed but also to result, is what is called Stochastic gradient descent, which looks at one rating at a time.

So the algorithm goes as follows, the details will be giving below:

For each rating $r(i,u)$:

- Calculate $e = r_{iu} - q_i p_u$ ($q_i p_u$ is the predicted rating, we will learn more about it below)
- Update x
 - $(24 \& x \leftarrow x - \alpha * e)$ (α is the learning rate)

You'd finish after one or more iterations over all the ratings.

So this is it for the math lesson, hope you learned something, and for sure it will be useful in the following. Stochastic Gradient Descent is used in the following.

11.6.4 And finally, to the Factorization

What if we try to disregard all the cells with nothing in it, and only calculate RMSE, looking only at the ratings we do know, and thereby avoiding the problem of the sparse matrix, and how to fill the empty cells. Don't get angry; we still need everything we talked about after the SVD. Trust me.

Our goal is to take all our ratings and create two matrices such that the i^{th} row of the item factor matrix multiplied by the u^{th} column of the user factor matrix will provide something that is close to the actual known ratings. Putting that into a least square problem we can say that we want to find matrixes Q and P which will minimize the following for all known ratings:

$$\min_{p,q} \sum_{(i,u) \in K} (r_{i,u} - q_i p_u)^2$$

where

q_i is the i^{th} row in the item factor matrix Q
 p_u is the u^{th} column in the user factor matrix P .

This can be done using the stochastic gradient descent algorithm described above. But basically, what we do is that for each rating we will update the two matrices Q and P or rather just a row and a column in the respective matrices. Before starting we need to decide how many features you want to end up with. In the FunkSVD we don't have the Sigma matrix to

calculate energy from. So, you will have to run it for several different number of features and see which is best.

So we can do the following algorithm:

For each f in features:

Continue until finished:

For each rating r_{ui} in ratings

- $e_{ui} = r_{ui} - q_i p_u$
- $q_i \leftarrow q_i + \gamma * (e_{ui} * p_u - \lambda * q_i)$ where
 - γ is the learning rate,
 - λ is called the regularization term.
- $p_u \leftarrow p_u + \gamma * (e_{ui} * q_i - \lambda * p_u)$ where
 - γ is the learning rate,
 - λ is called the regularization term.

I'll explain what this algorithm does and then show you how to implement it in Python. First, it calculates how much the predicted rating is off compared to the actual rating. This error is used to correct Q and P. With the error calculated; we will now update the two vectors q_i and p_u using the error we just calculated multiplied by the learning rate, which is often something like 0.001, and subtracting its own vector times a regularization factor. The last part is done to keep the length (values) of the vector small.

Now when you have run through all the known ratings once you will have two matrices that can be used to predict ratings. It is a good idea to shuffle the list of ratings before doing this algorithm because trends in the ratings might push the factorization in strange directions.

11.6.5 Adding Biases

Dot product of between the user and item factors

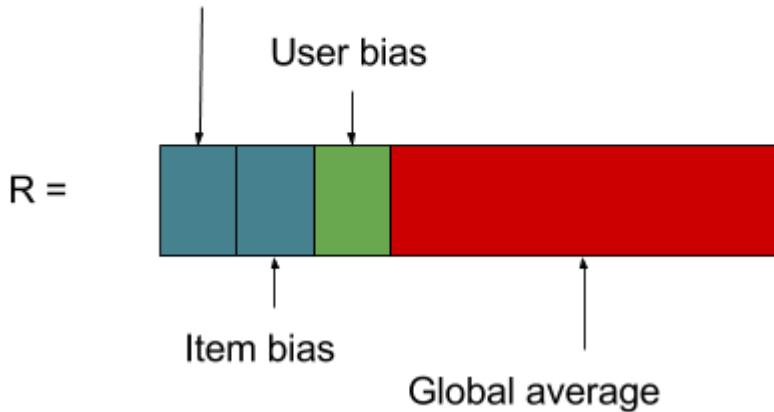


Figure 11.16 By now a predicted rating is a combination of many things

Above we talked about the biases, and even if the equation is already a bit complicated, it is worth adding them also. But what does it mean to have both user factors (a row in P) and bias? The way I think of it is that the act that user likes a specific type of films is encoded in the user factors, while the negative rater is encoded in the bias. A predicted rating is now the sum of four things as shown in figure 11.16

If we add those to the equation, the new function we want to minimize is the following

$$\min_{b,p,q} \sum_{(i,u) \in K} (r_{iu} - \mu - b_u - b_i - q_i p_u)^2$$

And now our algorithm will look like the following (this is an extension of the one above):

For each feature f :

Continue until finished:

- $e_{ui} = r_{ui} - q_i p_u$
- $b_u \leftarrow b_u + \gamma * (e_{ui} - \lambda * b_u)$
- $b_i \leftarrow b_i + \gamma * (e_{ui} - \lambda * b_i)$
- $q_i \leftarrow q_i + \gamma * (e_{ui} - p_u - \lambda * q_i)$

where

- γ is the learning rate,
- λ is called the regularization term.
- $p_u \leftarrow p_u + \gamma * (e_{ui} * q_i - \lambda * p_u)$

where

- γ is the learning rate,
- λ is called the regularization term.

Even if we are not going to drag too much through these equations, then it's worth knowing that we are doing it according to the stochastic gradient descent approach and the equations are found by taking the derivative of the squared error.

11.6.6 How to start and when to stop

So now that we have all the math and structure in place it's time to talk about the art of machine learning, because doing good machine learning is not just about knowing the math, it is also about understanding what how to initialize parameters, and what constants should be. The problem about arts are that they are hard to teach, because there are not a right and a wrong. Simon Funk was nice enough to say what values he gave the learning rate and the regularization parameter. Learning rate = 0.001, regularization = 0.02. With 40 factors.

But let's have a quick chat about how to figure good values for each of the parameters. The difference between what we are doing here and the evaluation we looked at in chapter 9 is that we are not just looking for getting more precision and recall, that is something we will look at when we are finished tuning the parameters, what we are concentrating on here is to teach the algorithm to understand the domain. That sounds pretty fancy, but let's just remember our goal, which is to provide recommendations for users online. How to do emulate that best, that is not by creating something that not only "remembers" the answers from the training data, but are also able to predict ratings from the test data. Basically, we want it to have little error both on the data it is training on and the one that we test it on. So when we calculate, or descent towards the two matrices for the user and item factors, we will use the training data to 'teach' about the domain, and the test data to know when it has extracted enough knowledge. If we teach it too little we will not have a result which 'understands' the domain to much it will be over-fitted on the training data, and instead of understanding the domain it has just mastered remembering the training data.

The tools we have to manipulate the algorithm is the following:

- initialize the factors? this decides where the gradient descend should start walking.
- learning rate? How fast we should move at each step.
- Regularization? How much should regularize the errors, should it be the same for all the expressions above, or should you split it into factors and bias.
- How many iterations? How specialized should it become to the training data.

There are many ways you can interpret what effect you would have on the result by setting these in different ways, the above is my way of doing it. But I would recommend you to work up an understanding of the effects of each of them. But also, to do grid search, which means that you try a lot of different values and see how it works. But you will be better equipped for your next recommender if you take the time to understand what power the parameters has.

The last sentence above should have troubled you a little bit. Because, the data we are using here, and most likely will not be the Netflix dataset which was used in the Netflix prize, so we should probably try out some different parameters just to see if we can make it work better. The following piece of code shows how we test out different number of parameters.

Listing 11.11: builder/matrix_factorization_calculator.py - Testing how many factors should be used.

```
def meta_parameter_train(self, ratings_df):
    for k in [5, 10, 15, 20, 30, 40, 50, 75, 100]:  
        self.initialize_factors(ratings_df, k)  
        test_data, train_data = self.split_data(10, ratings_df)  
  
        columns = ['user_id', 'movie_id', 'rating']  
        ratings = train_data[columns].as_matrix()  
        test = test_data[columns].as_matrix()  
  
        self.MAX_ITERATIONS = 100  
        iterations = 0  
        index_randomized = random.sample(range(0, len(ratings)),  
                                         (len(ratings) - 1))  
  
        for factor in range(k):  
            factor_iteration = 0  
            last_err = 0  
            iteration_err = sys.maxsize  
            finished = False  
  
            while not finished:  
                train_mse = self.stochastic_gradient_descent(factor,  
                                               index_randomized,  
                                               ratings)  
  
                iterations += 1  
  
                finished = self.finished(factor_iteration,  
                                         last_err,  
                                         iteration_err)  
                last_err = iteration_err  
                factor_iteration += 1  
  
            test_mse = self.calculate_mse(test, factor)
```

- ➊ try out the list of factors
- ➋ initialize all the factors and constants.
- ➌ split the data into training data and test data.
- ➍ When you want to speed up things, then use numpy arrays instead of pandas.

- 5 each factor is trained for 120 epochs that is running through the all the training ratings 120 times.
- 6 with stochastic gradient descent, it is important that you randomize the data.
- 7 we train one factors at the time.
- 8 continue until a termination function is true.
- 9 run stochastic gradient on all the training ratings, this will be shown in more detail below.
- 11 calculate mean square error (mse) on the test data.

Figure 11.17 shows the result of this, each factor run has been plotted

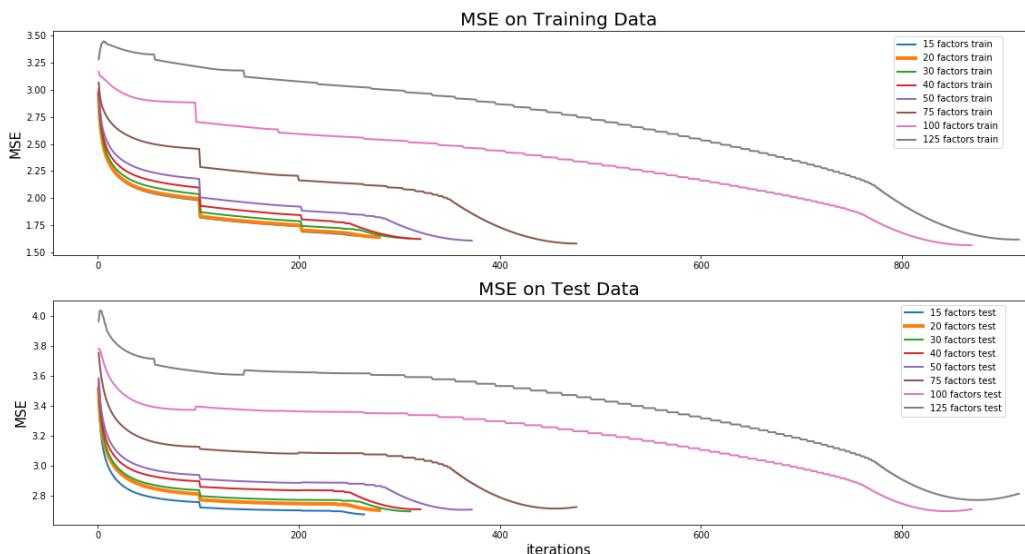


Figure 11.17 RMSE plotted for each iteration of the algorithm run. The top is the mean squared error on the training data, while bottom is the error on the test set.

After running the test above (it took around 12 hours on my MacBook w. 2.7 GHz Core i7 from 2016), I came to the conclusion that 40 factors. I also had to modify the number of iterations, Simon Funk wrote that he trained each feature for 120 iterations each, with this dataset I found that stopping it at either 100 iterations or when the last error was smaller than the current one. I think maybe it was worth giving it some slack, saying that if the algorithm has just started training a new factor, maybe you should leave it running for 10 iterations of so before stopping it. But I will leave that as a test for the reader. Simon Funk also mentions that he had different ideas of how to calculate the prediction error. There are so many ways you can combine it, and the best way to go about it is to come up with a hypothesis, and then do some test runs like above. Another way to look at the data above is to compare the error of the training compared to the test error. This is shown in figure 11.18. It is important that the lines have a more less 45% degrees such that test error is proportional to the training error.

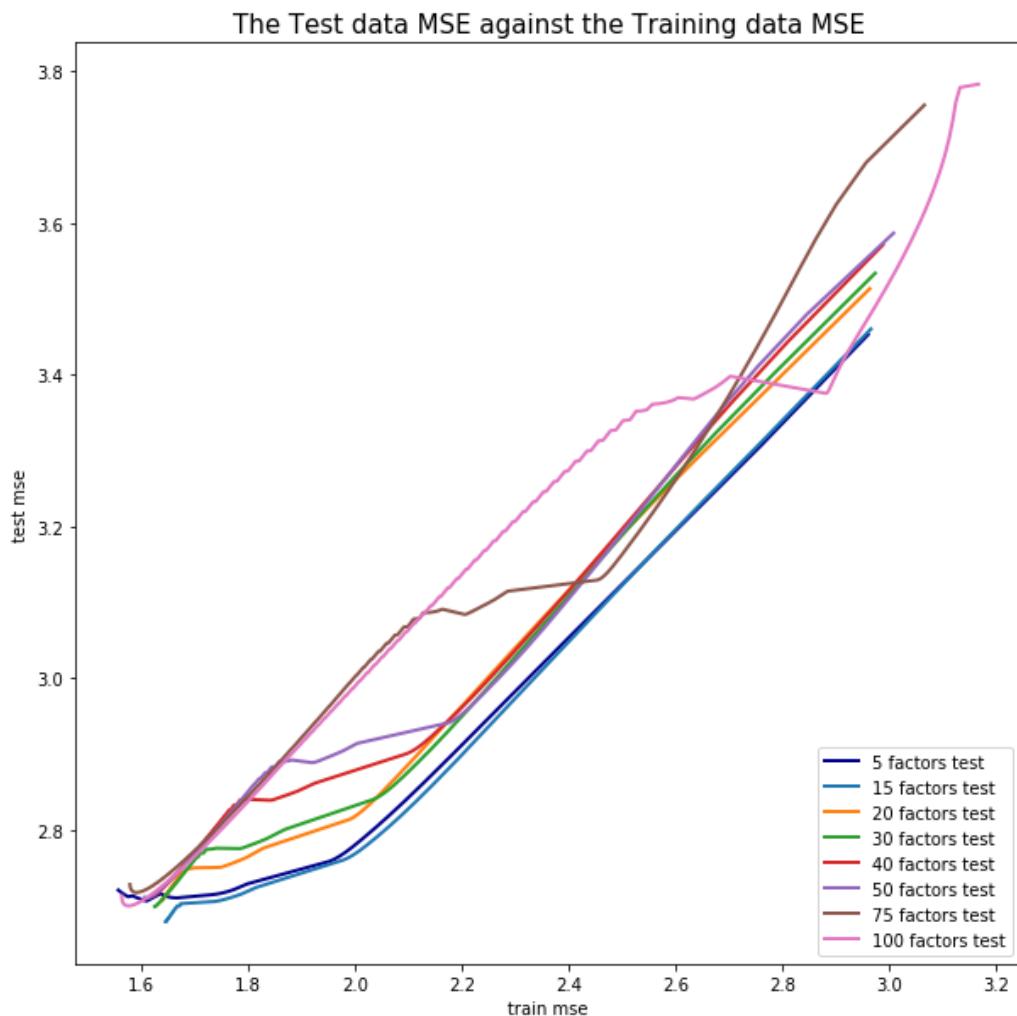


Figure 11.18 comparing test MSE with train MSE shows if you are starting to overfit

Many simply advice to run it for a certain number of times, like 50 or 100, while others recommend that you look at the MSE for each iteration and when the change in RMSE is below some value stop. Or the change made to the user and item factors. If you do plot the MSE as done in figure 11.18, it's a good idea to look for elbows in the chart. The elbow is usually where you stop fitting the algorithm to the known data and to start over fitting the algorithm, an example of one such can be seen in figure 11.18 line showing the training of factors 75. You can see the test MSE has a little elbow around 400 iterations. Here I was saying that you

should only use 20-factors. The 20-factor test line also has a very small elbow around 275 iterations, so maybe that would have been a good place to stop.

Overfitting

If your rating matrix is sparse, then you might run into overfitting problems, where the algorithm learns the training data too well, and suddenly the mse of the test data starts increasing, then you'd probably run into the problem of overfitting. Overfitting is when the matrices U and V can calculate precisely the right values for the existing ratings, but will be completely off when it comes to predicting new ones. A way to get around this, we introduce something call a regularization factor. So, what we will actually minimize is the following:

$$\min_{u,v} \sum_{(u,i) \in \text{known}} (r_{ui} - u_u v_i) + \lambda(\|u\|^2 + \|v\|^2)$$

The idea here is that we want the algorithm to find the best u's and v's, while not allowing any of them to become too big. The argument for doing this is not to let the factorization become too specialized towards the values in the training data that you are giving it, since we want it to be good at predicting ratings on new data. Overfitting is a very interesting topic that requires a lot more space than I have here. Have a look at the Wikipedia page: <https://en.wikipedia.org/wiki/Overfitting> for more info. In the funkSVD the fact that you deduct use the user factor to limit the item factor and the other way around, and the same with the biases, this should handle the overfitting problem.

11.7 Doing recommendations with FunkSVD

After all that work, we have four things:

1. Item factor matrix – where each column represents a content item, described by the latent factors that we have calculated
2. User factor matrix – where each column represents a user, described by the latent factors.
3. Item bias – some items are generally considered better than others, or worse. The bias describes the difference between the global mean and the items mean.
4. User bias – users have different rating scale; the user bias encompasses this.

With these, as we have seen we can calculate a predicted rating for any item for any user using the formula we talked about above:

$$\hat{r}_{u,i} = \mu + b_u + b_i + q_i p_u$$

And that is nice, because compared to the methods we have looked at before we are now able to provide predicted ratings on EVERYTHING!

Recommendations are more than just predicting ratings. We need to find a list of items that the user would rate high. We'll take a look at two ways of doing that: brute force and neighborhoods.

BRUTE FORCE RECOMMENDATION CALCULATION

Brute force is easy to describe. You calculate for each user for each item a predicted rating. Sort all the predictions and return the top N items. While you are at it you can also save all the predictions so that you have them ready when the user visits.

This is the no-nonsense way of doing stuff, the thing to remember is that it might take some time to do, and require your system to do a lot of calculations that might never be used for anything. Maybe we can optimize it a bit. Instead we will save the factors and the biases and use those to calculate the recommendations.

NEIGHBORHOODS RECOMMENDATION CALCULATION

I spend a lot of time describing neighborhoods in chapter 8, so I won't drag you through that again. The only thing that has changed is that instead of using the original rating data, then we will use the factors that we calculated. It means that we are calculating similarities somewhere where things are closer and in a smaller dimension, which makes it easier.

If we look at the factor space from above, shown again in figure 11.18. We can create recommendations in two ways:

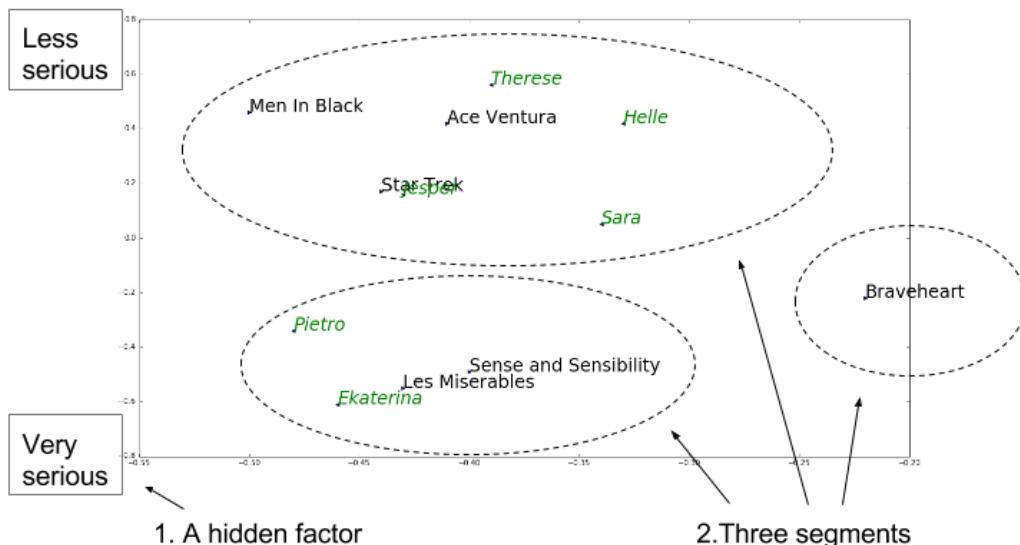


Figure 11.19 coordinate system showing one interpretation of how the hidden space could be interpreted. Here we can see that Ekaterina and Pietro likes Sense and Sensibility and Les Miserables. But no one likes Braveheart.

User vector

A way to create recommendations is to simply find the items with a factor vectors which are close to the active users vector. They are all in the same space, so why not. But if you do that, then you should consider that the user is placed in the middle between all the items the user likes, that means that if the user likes only one type of movies, that it is fine. If you are like me and like Italian dramas and superhero movies then the neighborhood around you, will be in between everything you like, and you might not like any of the things close to you.

The items the user likes

So, we are back with the square from chapter 9 also shown in figure 11.21. going through all the items that the user has rated positively and finding similar items. Or finding similar users and recommend items they liked.

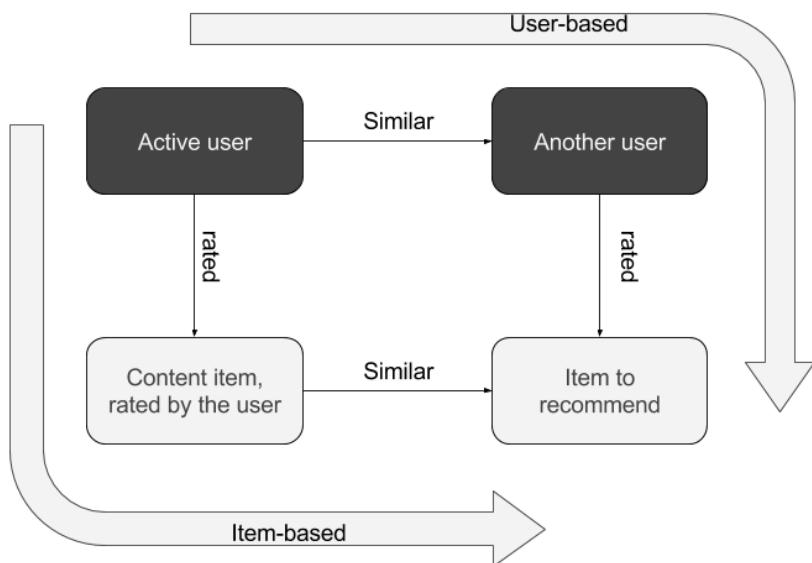


Figure 11.20 The different ways to go from active user to recommendations. Either by finding similar users or look that the active user's items, and find similar.

11.8 Funk SVD implementation in MovieGEEKs

We have had a few implementations by now, so what we want to look at here is, of course, the recommendations, but also how to verify if the model is working.

To train the algorithm, you will need to run `builder/MatrixFactorizationCalculator.py`. What happens inside there will be described in the following:

TRAINING PHASE

Initialization of all the biases and factors, and retrieve the ratings. Since we will run through the ratings a few times, it's better to load them up front and keep them in memory.

Listing 11.13: builder\matrix_factorization_calculator – initializing the factors

```
def initialize_factors(self, ratings, k=25):
    self.user_ids = set(ratings['user_id'].values) ①
    self.movie_ids = set(ratings['movie_id'].values) ②

    self.u_inx = {r: i for i, r in enumerate(self.user_ids)} ③
    self.i_inx = {r: i for i, r in enumerate(self.movie_ids)} ④

    self.item_factors = np.full((len(self.i_inx), k), 0.1) ⑤
    self.user_factors = np.full((len(self.u_inx), k), 0.1) ⑤

    self.all_movies_mean = self.calculate_all_movies_mean(ratings) ⑥
    self.user_bias = defaultdict(lambda: 0) ⑦
    self.item_bias = defaultdict(lambda: 0) ⑦
```

- ① create a set of all user ids
- ② create a set of all movie ids
- ③ create a dictionary from user_ids to a number, such that we can use numpy arrays instead of pandas. We do that to make it faster
- ④ create a dictionary from movie_ids to a number, such that we can use numpy arrays instead of pandas. We do that to make it faster
- ⑤ create two matrices of factors, initialized all with 0.1.
- ⑥ calculate the average of all the movie ratings.

To test the error between the actual ratings and the calculated one, we need a predict rating method. The method is identical to the prediction method which will be used in the end.

Listing 11.14: builder\matrix_factorization_calculator – predict method

```
def predict(self, user, item):
    avg = self.all_movies_mean
    pq = np.dot(self.item_factors[item], self.user_factors[user].T) ①
    b_ui = avg + self.user_bias[user] + self.item_bias[item] ②
    prediction = b_ui + pq ③

    if prediction > 10:
        prediction = 10 ④
    elif prediction < 1:
        prediction = 1
    return prediction
```

- ① calculate the dot product of the current users factors and the current item.
- ② sum the biases
- ③ make sure that the prediction is between 1 and 10.

With these we are ready for the actual training algorithm, it's the same as described above.

Listing 11.15: builder\matrix_factorization_calculator.py

```

def train(self, ratings_df, k=20):

    self.initialize_factors(ratings_df, k)
    ratings = ratings_df[['user_id', 'movie_id', 'rating']].as_matrix() ①
    ②

    index_randomized = random.sample(range(0, len(ratings)),
                                      (len(ratings) - 1)) ③

    for factor in range(k):
        iterations = 0
        last_err = 0
        iteration_err = sys.maxsize
        finished = False

        while not finished:
            start_time = datetime.now()
            iteration_err = self.stochastic_gradient_descent(factor,
                                                               index_randomized,
                                                               ratings)

            iterations += 1
            finished = self.finished(iterations,
                                      last_err,
                                      iteration_err)
            last_err = iteration_err
            self.save(factor, finished) ④

```

HOW MANY ITERATIONS ARE NEEDED TO RUN IT

As we talked about already, there is no easy way to decide when to finish. Here is the one I used training it.**Listing 11.16: builder\matrix_factorization_calculator.py – finished?**

```

def finished(self, iterations, last_err, current_err):

    if iterations >= 100 or last_err < current_err: ①
        print('Finish w iterations: {}, last_err: {}, current_err {}'
              .format(iterations, last_err, current_err))
        return True
    else: ②
        self.iterations +=1
        return False

```

- ① if more than 30 iterations has passed or the difference between the error has gone below 1, then its time to stop, and move on to the next factor.
- ② moving on, just incrementing the number of iterations.

SAVING THE MODEL

We wouldn't want all the work to be lost, so we will save the model to some files. Now saving the model to Json is just one way of doing it, and depends on how you want to use it. I have chosen to save each of the factor matrixes in a file, as well as the biases.

Listing 11.17: builder\matrix_factorization_calculator.py – storing the factors

```
def save(self):
    print("saving factors")
    with open('user_factors.json', 'w') as outfile:
        json.dump(self.user_factors, outfile)
    with open('item_factors.json', 'w') as outfile:
        json.dump(self.item_factors, outfile)
    with open('user_bias.json', 'w') as outfile:
        json.dump(self.user_bias, outfile)
    with open('item_bias.json', 'w') as outfile:
        json.dump(self.item_bias, outfile)
```

Json in a file is not that fast to retrieve, and we want to be fast. So, it depends on what you want to do now. If you want to do the brute force, and save all the predicted ratings in the database, then you need to take the dot product between all users and all items. Save the ones that are above a certain threshold in the recs database and then do a simple look up when needed.

The other way is to calculate similarities using cosine similarity as we did in chapter 8. And calculate the ratings the same was as we did then. Here we do a slower version, which loads the files every time it needs to do a recommendation.

ONLINE PHASE

The following shows how to do recommendations by calculating predictions and then ordering them according to the highest prediction.**Listing 11.18:**
recs/funksvd_recommender.py

```
def recommend_items_by_ratings(self, user_id, active_user_items, num=6):
    rated_movies = set(active_user_items.values('movie_id')) ①
    user = self.user_factors.loc[user_id] ②

    scores = self.item_factors.dot(user) ③
    scores.sort_values(inplace=True, ascending=False) ④
    result = scores[:num + len(rated_movies)] ⑤

    recs = {r[0]: r[1] + self.user_bias[user_id] + self.item_bias[r[0]] ⑥
            for r in zip(result.index, result) if r[0] not in rated_movies}

    sorted_items = sorted(recs.items(),
                          key=lambda item: -float(item[1]['prediction']))[:num] ⑦

    return sorted_items
```

- ① Get the id's of all the movies the user has already rated. We don't want to show them again.
- ② the user_factors are loaded from the files we saved during training.
- ③ the item_factors are loaded from the file we saved during training
- ④ Sort the ratings in descending order
- ⑤ Take the head of the list containing only enough elements for the number of elements we want to return, but leaving room for the case where all the users rated items are there.
- ⑥ Create a dictionary, content id – predicted rating.
- ⑦ Create the data format the frontend expects and cut off the number that should be returned.

To do a manual test and check a recommendation to somebody, have a look at this user id 400006 which has a taste profile like this:



Figure 11.21 taste profile of user 400005 (<http://localhost:8001/analytics/user/400005/>)

This user receives the following recommendations (screen dump from the analytics part of MovieGEEKs <http://localhost:8001/analytics/user/400005/>) in figure 11.23

Personalised recommendations

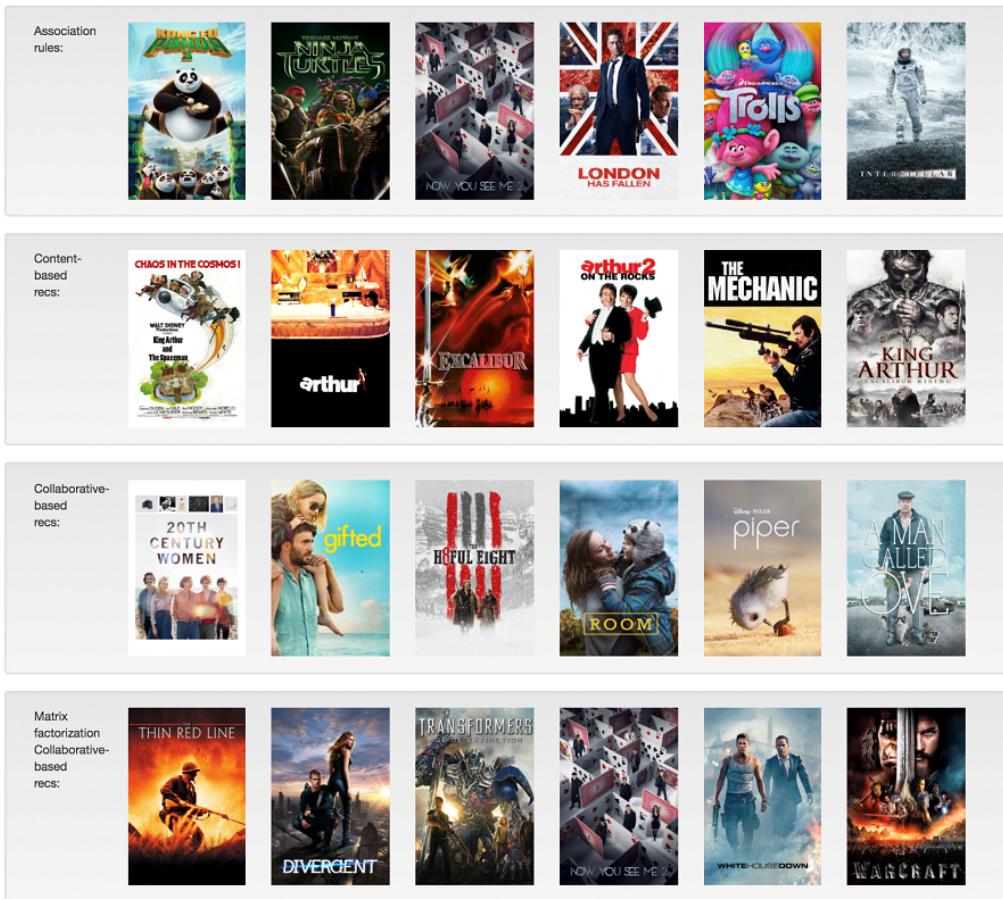


Figure 11.22 The recommendations we have calculated so far. The Association rules, then the content based and collaborative filtering and finally the matrix factorization recommendations. I think the recommended movies are quite action packed, but they fit the movies the user has watched quite well. But it is of course a subject opinion.

The problem with looking at one user's recommendation is that it could potentially be one out of only a few where it provides good recommendations, we can never be sure. That's why the only real way to see if the model is good is by putting it in production.

11.8.1 What to do with outliers

In one of the first iterations of the recommender, I had just built one of the first models, and I thought now everything was ready for sendoff. I came across a user who liked cartoons like

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/practical-recommender-systems>

Licensed to Asif Qamar <asif@asifqamar.com>

Kung Fu Panda and *LEGO Batman* and but the funkSVD produced the recommendations shown in figure 11.23

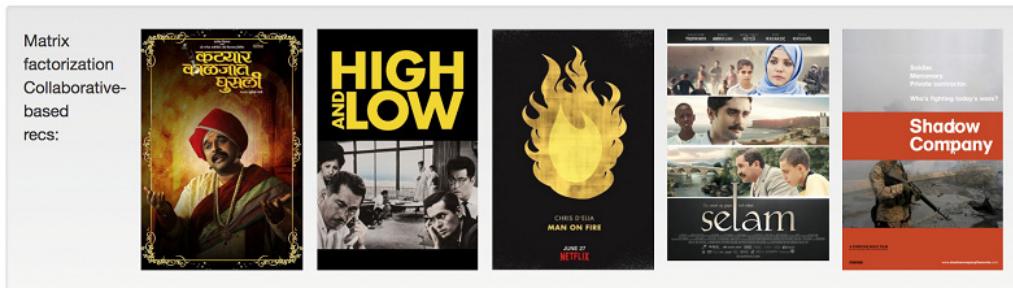


Figure 11.23 Some outliers have sneaked into the recommendations

It turns out that these are actually not very related to neither *Kung Fu Panda* or *LEGO Batman*. The user taste shows a liking to Adventure, Animation, Crime, Fantasy and Thriller. A little bit more poking around showed that they were 5 out of the 5 items which had an item bias above 5. The algorithm does the dot product between the user and all the item factors, and to this the item biases are added before we order them and take from the top. I am a bit torn about whether you should order the items before adding the item biases or not.

If you change the algorithm to sort the items before adding the item bias then they will still boost the recommendations, but only within the list of items which are similar to the factors that the user like. Doing this changed the recommendations into what is shown in figure 11.24



Figure 11.24 Adding the bias after ordering the recommendations, shows better recommendations.

Another thing I played around with was to reduce the learning rate of the bias' which also helped with this problem, since it made the bias' being smaller, and more diversity into the user and item factors. It's a matter of taste what is better, and something that can give you many sleepless nights. It is difficult to say what to say are the best settings, you should therefore play around with it.

11.8.2 Keeping the model up to date.

This model we created quickly goes out of date, so it should be recalculated at least daily. Ideally, you should start a new run as soon as it is finished, but depending on how much your data changes you can do it ones a night or even once a week. But remember that every time a user interacts with a new item, it might be evidence that will show connections between items and give inklings about the user's taste.

11.8.3 Faster implementation.

The above matrix factorization is done using gradient descent, if you have many millions of products and users, that might become a quite slow process. There is an alternative way which is called Alternating least squares (ALS) which is not as precise as the one above, but should work fairly well anyway.

11.9 Explicit vs implicit data.

If you have a dataset of implicit data, you could do as we did in chapter 4 and deduce ratings from it, alternatively you can use it directly, by slightly changing the algorithm above, for more information on that you should find the article *Collaborative Filtering for Implicit Feedback Datasets* by Yifan Hu et al. If you want to skip directly to implementation of it have a look at the implicit⁹¹ framework is a fast implementation for collaborative filtering for implicit data. I have heard that it is used in production environment where there is a lot of data. I can tell you where, but then I am afraid I have to kill you afterwards.

As mentioned, the implementation uses slightly different algorithms than described here. So, it can be a great exercise to look through the code to understand alternative ways to do it.

11.10 Evaluation

When you work with FunkSVD or other machine learning algorithms, you usually have many parameters to tune before starting the actual evaluation. Above we have talked a lot about the error of the training data and compare it with the error on the test data. When you are finished with adjusting those variables you can do the evaluation. Some would say that it is the same thing. But you probably don't want to find hyper-parameters doing cross validation simply because it takes too long. You will almost always use cross validation to evaluate the performance of the algorithm. The principle is exactly the same as the one described in chapter 9.

⁹¹ <https://github.com/benfred/implicit>

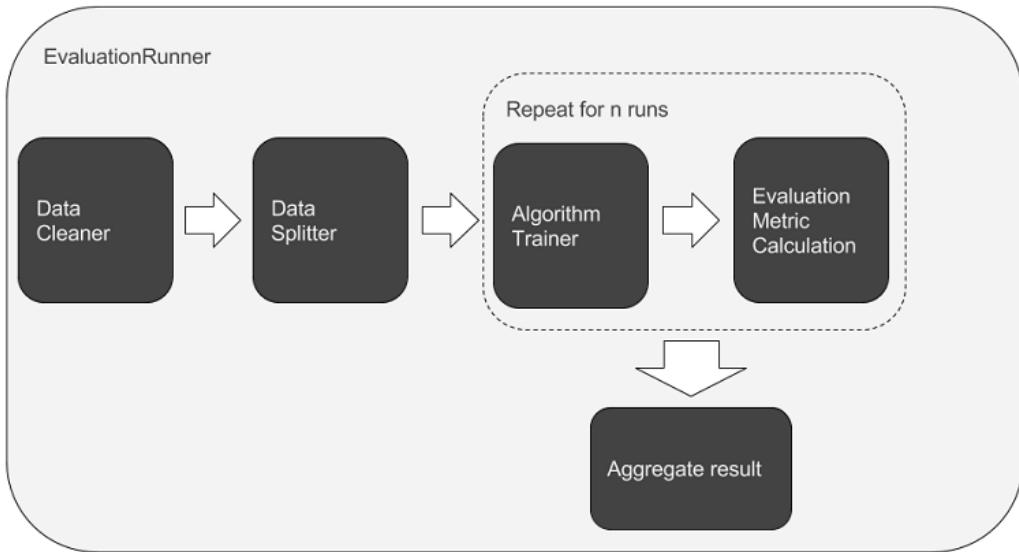


Figure 11.25 A diagram showing how the Evaluation runner works

You clean the data, so you only have relevant information, then you split the users into k folds. Then for each of these folds, you hold one out, and split it into train and test data for the users in that fold. The training data from the fold is merged/appended to the k-1 other folds, and used to train the algorithm, then the algorithm is tested using one of the Metrics described in Chapter 8, and then finally you aggregate the result.

You can run the evaluation by executing the following:

Listing 11.19: executing the evaluation.

```
> python -m evaluator.evaluation_runner -funk
```

It will create a csv file with the data used to show the evaluations. I would encourage you to check the code and see how it is done. The script does not run cross validation out of the box, you need to tweak the code to do it. It is said to 0 folds, which means no cross validation.

I have run an evaluation on the model with 40 factors calculating the precision and recall as we have done in the two previous chapters, the result is shown in figure 11.25.

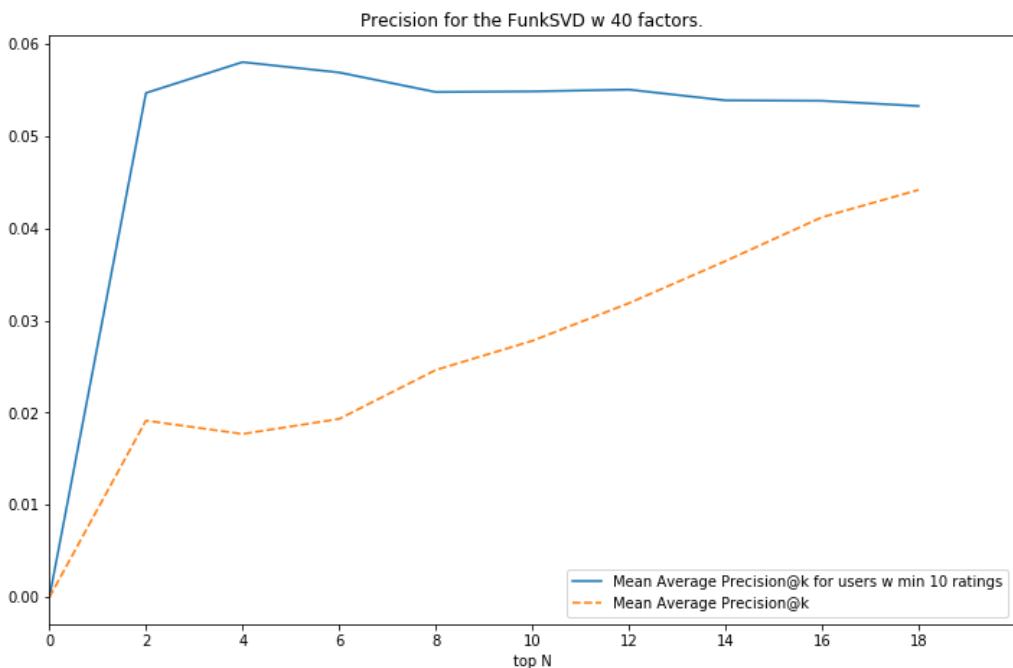


Figure 11.26 Mean Average Precision of the funkSVD, the bottom line is when we test with users with only one rating, while the upper one is we restrict the test to have at least 10 ratings.

Looking at the result of the matrix factorization I think that it is great to see that it can service 100% of the users, but I am also a bit disappointed to see that it only recommends 0.11% of the items, which is a bit more than 3200 items. One reason for that could be that we should add more factors to allow for more diverse taste profiles. I tweaked the bias learning rate a bit from 0.0005 which produced a model which only covered around 200 items, to 0.002 which now recommends more than 3200 items. There are still a lot missing to cover the full set of 28700 items. But again, it is something to play with.

And a great exercise for you the reader to try out. The precision is much better than the content-based algorithm, where both recall and precision were more than half this. The thing that you might make a fuzz about is the fact the neighborhood model gets a better recall than this one. But the difference here is that we have a 100 % user coverage. That means that users who only rated one item is also included while we took those away from the data before did the neighborhood model. If we cleaned out all users with few ratings at as we had to in CH 8, then we would probably have a superior precision and recall here.

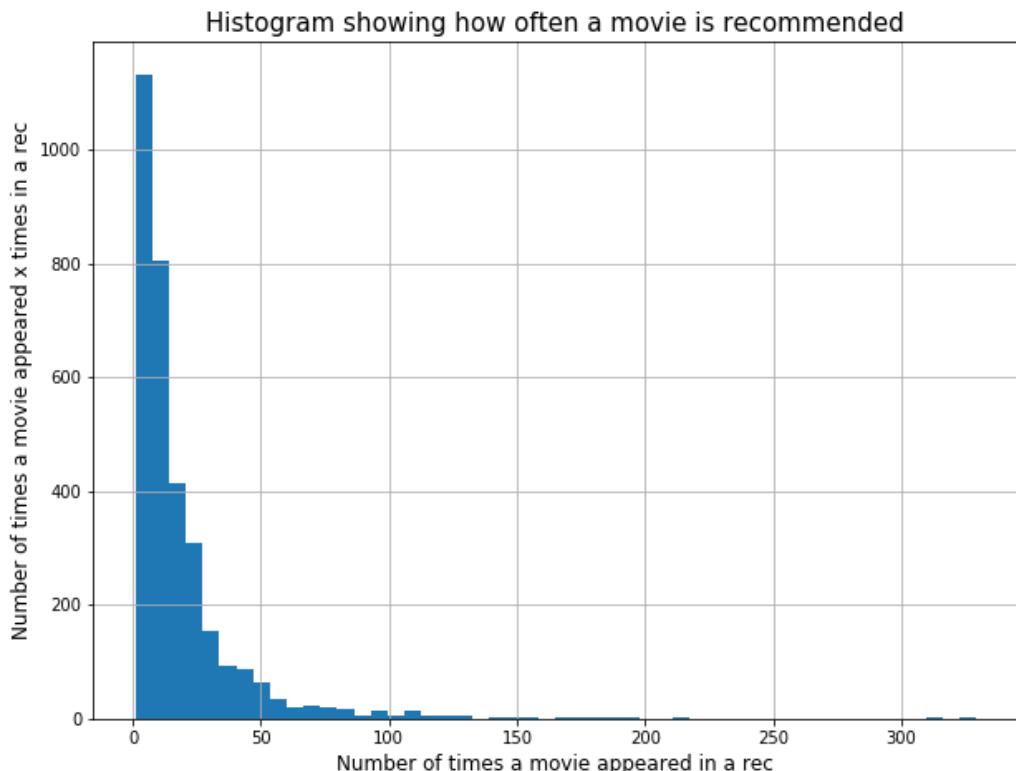


Figure 11.27 Show how many movies are only shown a certain number of times. For example more than 1000 movies are only shown to one user.

11.11 Levers to fiddle with for the FunkSVD

There are so many levers and configurations that can be changed and tweak when you do this algorithm, and what makes it more complicated is that they are all depend, meaning that you can't expect that optimizing one parameter, will always be the best possible value, if you then change another factor. So you should try all combinations to be completely sure which is best. That can be solved with gridsearch⁹², or something similar.

What ratings to add to the training, the users that has only done one rating, doesn't actually help your collaborative filtering algorithm much, so they could be good to take them out. The only thing they help is that they add to average rating of the movies.

⁹² http://scikit-learn.org/stable/modules/grid_search.html

Item and User Factors are representations of users and items using the latent factors which we have been training. It is good to check that factors are not too large, I would be worried if any vector had values wasn't around 1. You can tune this by regularization, the larger the regularization the more the factors are kept close to 0.

Bias initialization matters, as they will determine how big the prediction error will be on the factors in the beginning, these you also want to not too large. Consider that if you have the average rating of all the movies, then adding the user bias and item bias should not bring it outside of the rating scale. The dataset we are using here use a rating scale from 1-10, the average rating is around 7 so and item bias and user bias shouldn't be too much above 1.5 each.

Determining the right number of iterations is also a matter of taste. If you let it run through too many iterations on the early factors, you risk that all signal is pushed into the first dimension, and the following will only have little signal. Too few iterations and the vectors will never align correctly. Based on how many iterations you need to decide the Learning rate of the factors and the bias learning rate, too large and it might always overshoot the optimum, too small it will never move from its initial position. This is a very short listing of things to play around with. I would say that there are many more. Remember to have a good measure of quality then you test it. Getting a low RMSE/MSE might not actually result in the best recommendations, but it how to do a function which actually captures that. But again, remember make a hypothesis, be sure you know how to measure the result, and make the testing as easy as possible.

11.12 Summary

This has been quite a complex chapter, and we have learned a lot in this chapter, below is the list. Matrix factorization is a topic in itself, and here we really just scratched the surface. But if you understood this chapter you should be able to build on it and make much more complex things. In this chapter, we looked at:

- SVD is a way of doing matrix factorization; it is good and well accepted in many libraries out there, which makes it a good method to use, the drop side is the fact that it won't work if your matrix isn't complete, i.e., with no empty cells. So, you will have to fill them in with something. That something is hard to do since they are empty because we don't know what should be there.
- Baseline predictors - One way to fill out those empty cells are baseline predictors, which indicates the difference for users from the global mean, and same for items.
- A way that allows for us to use the sparse matrix of ratings is to use the funky method first tried by Simon Funk.
- To train the Funk SVD we also looked at gradient descent and stochastic gradient descent, which is a super tool to solve optimization problems like the one we defined to optimize the Funk SVD.

12

Taking the best of all algorithms – implementing hybrid recommenders

This chapter is a hybrid of many sections, you will:

- You'll learn to combine different recommenders to take advantage of the strengths and weaknesses of different types of recommender systems
- You'll have a tour of the different overall classes of hybrid recommenders.
- You'll get introduced to ensemble recommenders
- Having knowledge of ensemble recommenders, you will look at how to implement a specific algorithm called Feature-Weighted Linear Stacking

A Toyota Prius is a hybrid car because it utilizes two different engine technologies: the combustion engine and a battery-powered engine. It is supposedly one of the most energy efficient cars ever made⁹³, even though Toyota did a lot of innovation, at its core, it used two well-known technologies. Hybrid recommenders are basically the same idea; you combine recommender algorithms to get a more powerful tool. They not only improve the average result but also attempt to mitigate the corner cases where algorithms do not really work.

⁹³ Got this from https://en.wikipedia.org/wiki/Toyota_Prius

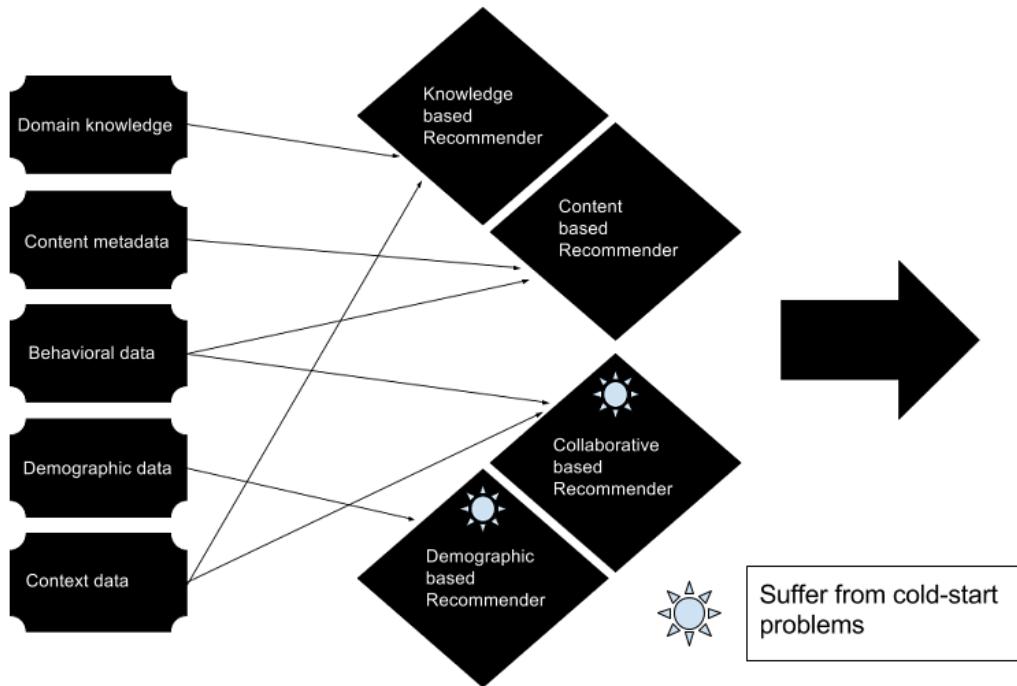


Figure 12.1 Illustrates four commonly recognized classes of recommenders and the type of data they consume

Figure 12.1 shows the four most acknowledged classes of recommender systems and their data sources. So far we have talked about each recommender system as something that runs alone and in a silo, but the world is far from this ordered. To be able to provide recommendations always you need to do a mix or a hybrid between more than one. Also, if you have access to more than just one of the data sources shown in the figure, it's a sin not to use all of them!

There are no limits to how you can combine algorithms, and if this chapter explained all of them, it would quickly become a book in itself. To avoid doing that, we will start out with a small survey of the different categories that are commonly considered. Then we will take a closer look at the details for the feature-weighted linear scaling algorithm. Finally, we will see how to implement it in MovieGEEKs.

12.1 The confused world of hybrids

Here is how I like to look at hybrids. “Hybrid recommenders” is a common term for the different types of recommenders shown in figure 12.2.

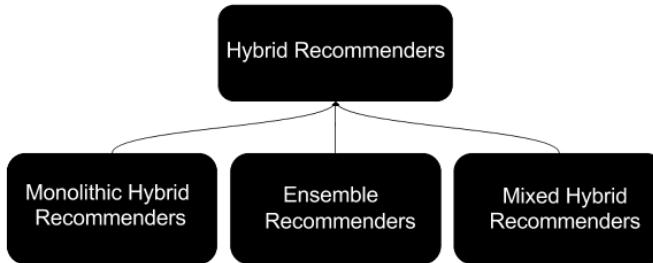


Figure 12.2 There are overall three different types of hybrid recommenders. Monolithic's steal parts from different recommenders and makes it into one. Ensemble uses the output of several to come up with the recommendations, while mixed are simply switching between using different recommenders.

The monolithic takes components of various recommenders and glues them together in new ways, the ensemble runs different recommenders and then combines the result into one recommendation, while a mixed recommender runs a number of recommenders and return all of them. Each of these will be described in detail below.

12.2 The Monolithic

If you look up the word *Monolithic*, you don't get a picture of something cutting-edge. Instead, it means something that is made out of just one piece of stone, like the heads on the Easter Islands, or refers to organizations that are powerful, rigid and slow to change. So, while *monolithic* is probably not something you want to put in an ad for your company, nevertheless it is what these recommenders are called.

In our case, a monolithic hybrid recommender is defined as the Frankenstein of recommenders, it contains parts from different types of recommender algorithms. A recommender contains many different components; it could contain *Item Similarity*, *Candidate Selection* and *Rating Prediction*, to name a few. A monolithic recommender can mix components from different recommenders or even add new steps to improve overall performance. Figure 12.3 illustrates a monolithic recommender which uses the item similarity part of Recommender 1 and candidate selection and prediction from Recommender 2.

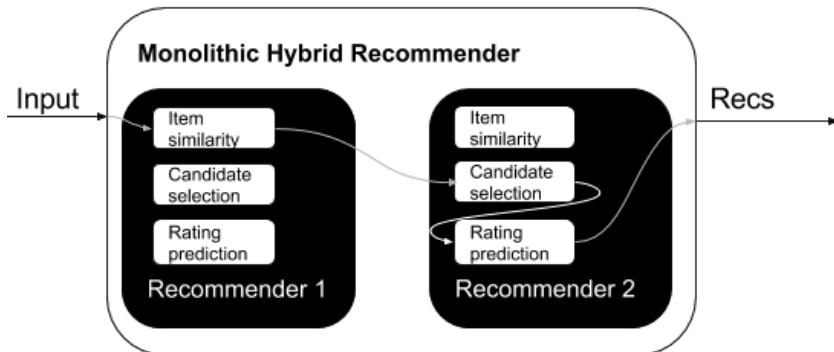


Figure 12.3 A monolithic hybrid recommender is composed of other recommender system parts.

As in the figure you could say that instead of finding similar items by finding items that has been rated by the same users as we saw in chapter 8 on Collaborative filtering, we could use a content based approach and find all the items that are similar content-wise, and then use those in a collaborative filtering approach, and calculate predicted ratings the same way as in collaborative filtering. The possibilities are endless. A monolithic recommender can also add another step as described in the following.

12.2.1 Mixing features from content-based features with behavioral data to improve collaborative filtering recommenders.

The whole point of hybrids is to take advantage of more types of data. An example of a monolithic hybrid recommender could be a collaborative filtering recommender with one extra pre-processing step which adds ratings to the rating-matrix such that the collaborative filtering will connect things that are related content-wise.

Let's take an example. Say that we have a list of films all the genre Sci-Fi, like:

Table 12.1 An example of a rating matrix

	Sci-fi 1	Sci-fi 2	Sci-fi 3	Sci-fi 4
User 1	4	4		
User 2	5	4		
User 3			2	4

Using neighbourhood collaborative filtering on the rating matrix in table 12.1, we would not get any similarity between the first two and the last two, so we wouldn't be able to recommend anything to any of the users as there are no connections to unseen films. So what we need is a user to bridge the content. Since we know it is all sci-fi films then it might be

good to add a user into the matrix which loves all sci-fi films. We could update the rating matrix like in table 12.2

Table 12.2 An example of a rating matrix

	Sci-fi 1	Sci-fi 2	Sci-fi 3	Sci-fi 4
User 1	4	4		
User 2	5	4		
User 3			2	4
Sci-fi lover	5	5	5	5

This would enable us to link sci-fi films together, and thereby also recommend films of the same genre. We could also go the next step and use the LDA model we implemented in chapter 10 and make Pseudo-user for each of the hidden topics and then add those connections to the mix.

There are some more technical details to understand, if this is a way you want to go, read Melville and colleagues article called "Content-Boosted Collaborative Filtering for Improved Recommendations"⁹⁴.

The example above will require some work, and monolithic recommenders, in general, will probably require some work to change current recommenders to work in a hybrid, therefore if you already have recommenders in place, you might want to try out mixed hybrids or ensembles.

12.3 Mixed Hybrid Recommender

A mixed hybrid actually doesn't do much mixing, a mixed hybrid simply returns the union of all the results.

A way of using a mixed hybrid that I have used myself is to simply have a hierarchy of recommenders. Consider the recommender along a scale of personalization, make the first one as personalized as possible and then continue until you are using item popularity to give recommendations. You would do this because often the most personalized recommender only produced one or two recommendations, then the next recommender would produce some more, and in that way you will always have a good quantity of recommendations but with as much quality as possible. Figure 12.4 illustrates a mixed hybrid recommender.

⁹⁴ <http://www.cs.utexas.edu/~ai-lab/pubs/cbcf-AAAI-02.pdf>

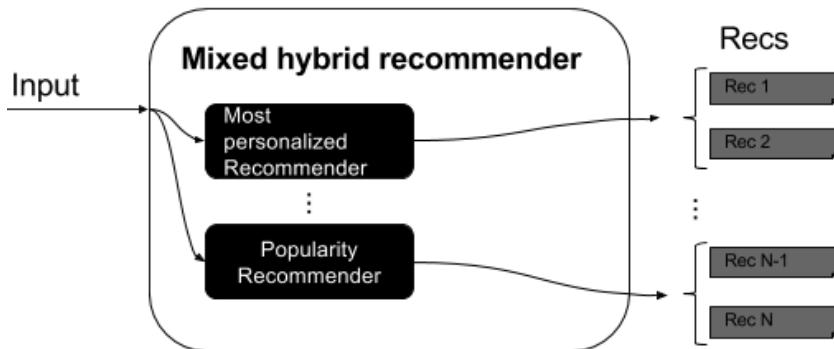


Figure 12.4 Mixed hybrid recommender, which takes all from the most personalized recommender, then the next more personalized, etc.

If you have several good recommenders, and each recommender returns a score then you can return the list ordered accordingly, only you need to remember that the scores should be normalized so all results are on the same scale. Keeping with the idea of having several recommenders running we now turn to ensembles.

12.4 The Ensemble

An ensemble is defined as a group of things viewed as a whole rather than individually. It is the same for Ensemble recommenders: we combine predictions from different recommenders into one recommendation.

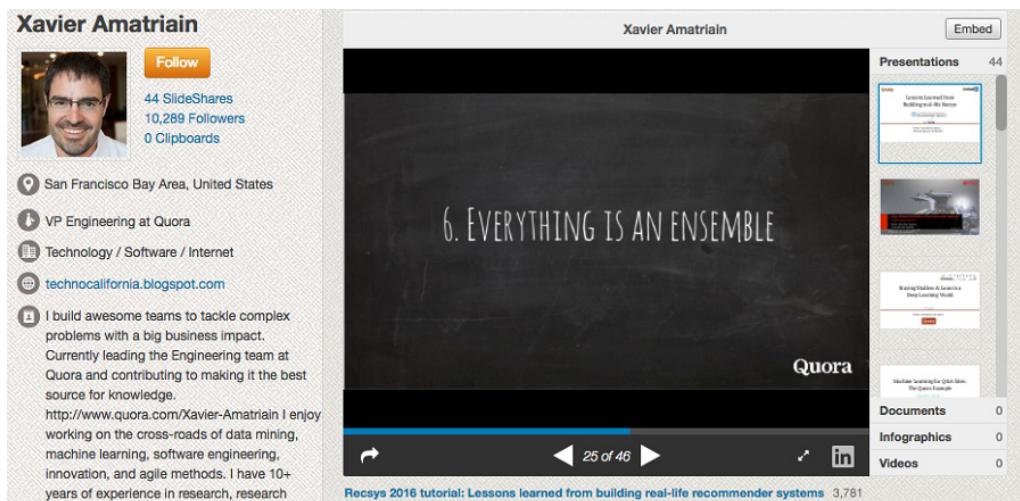


Figure 12.5 One of the takeaways from RecSys2016 was that everything is an ensemble.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/practical-recommender-systems>

Licensed to Asif Qamar <asif@asifqamar.com>

One of the takeaways from RecSys 2016 – the annual recommender systems conference in 2016 was that if you are a start-up and want to get into recommender systems, you should start out with matrix factorization (the topic of chapter 11) and then from there just add recommenders to create an ensemble recommender. So, keep reading, and afterwards go and check out Xavier's slides, they contain more relevant advice.⁹⁵

If you have two recommenders running already, let's say content based and collaborative filtering, then why not run them contemporary as seen in figure 12.6, then combine the result to try to get an even better result?

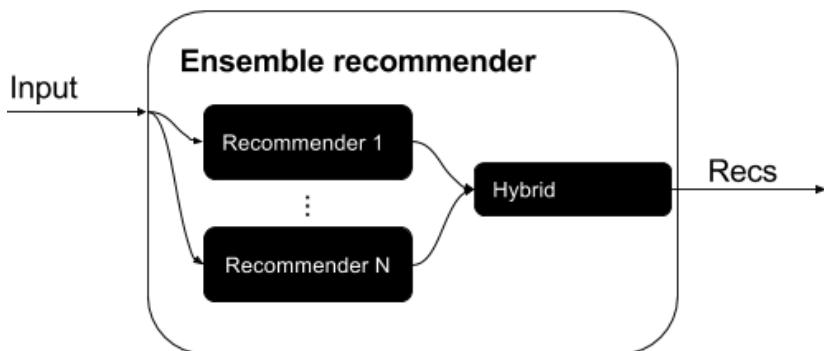


Figure 12.6 An Ensemble hybrid runs a number of recommenders and then combines the result before delivering it as one recommendation

The idea with the ensemble is to calculate recommendations using several full recommenders. And then combine them somehow. There are different ways you can take the result of a number of recommenders and make it into one. You can simply do a majority voting saying the objects that occur most is the top one, then the next one and so on. The difference between an ensemble and a mixed recommender above, is that the hybrid might not show anything of the result that one recommender did, while the mixed hybrid always shows everything. it's not a huge difference really.

⁹⁵ <http://www.slideshare.net/xamat>

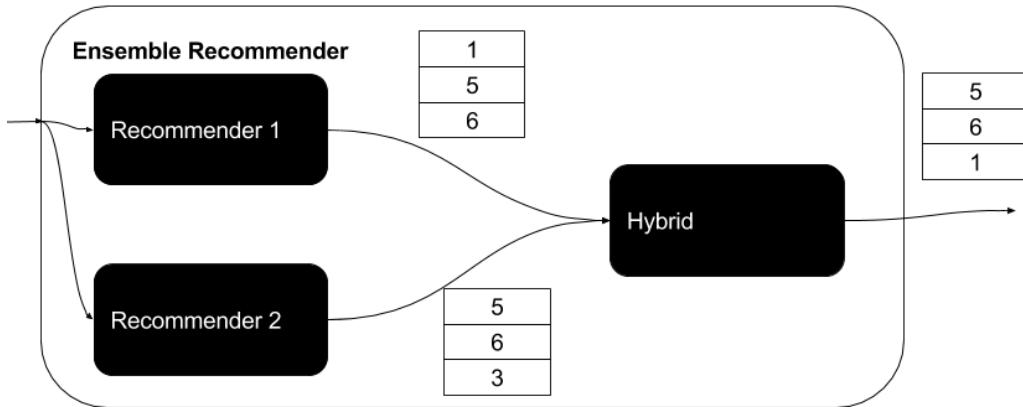


Figure 12.7 Example of how an Ensemble Recommender might work

Figure 12.7 shows an example of how an ensemble recommender might work. Recommender 1 returns a recommendation top 3 containing movie 1, 5, 6. Recommender 2 returns 5, 6, 3. Then the hybrid would return 5, 6, 1, of course depending on how you calculate ties. But 5 because it occurs twice, one in each result. Then 6 because it appears in both too. Then 1 because it appears as the first in recommender 1. And 3 will be dropped.

Often you will hear talk of switched or weighted ensembles. Which we will look at in the following.

12.4.1 Switched ensemble recommender

A switched ensemble recommender is about using the best tool for the job. So if you have two or more recommenders then a switched ensemble recommender will decide which of them to use given the context of the request. For example, you might have two different recommenders for two different countries, and when a user comes in from one country the result of one recommender is shown, while if somebody from the second country enters it's the other recommender that produces the output.

It can also be switched on time of day, maybe one works in the mornings, while another is strong in the evening.

On a newspaper, you could imagine that the national news section should be filled with latest news, while the culture page on books might be more about content-based recommendations. In other words, you switch recommenders based on which section of the site a user is on.

In its simplest form it could be one where the switch could be between users who rated less than 20 movies and the ones who did as shown in figure 12.8. Users with more than 20 ratings receive output from collaborative filtering recommender, while users with less from the Shopping basket recommender.

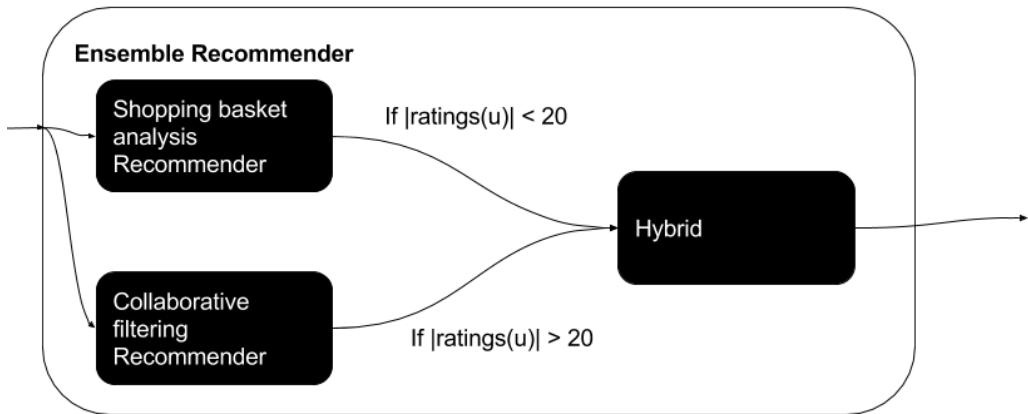


Figure 12.8 Example of a switching ensemble. Users with less than 20 ratings receive output from collaborative filtering recommender, while users with more than 20 will receive recs from the basket-analysis algorithm (see Chapter 6).

If a user is logged in, you know more about the user. That might be a good reason to use a different recommender than you'd use if the user is not logged in. If you instead want to combine forces of the different recommenders, read on and learn about weighted feature ensembles.

12.4.2 Weighted Ensemble Recommender

Consider two algorithms we looked at earlier in this book, collaborative filtering (we don't care if we are talking about the neighbourhood or SVD type now) and content-based filtering.

Content-based filtering is good at finding similar content. If we know a user likes a topic, we can use content-based filtering to find similar things. The problem is that content-based filtering doesn't distinguish between good and bad quality; it is only concerned with topic or keyword overlap. On the other hand, collaborative filtering doesn't really put any importance into items being about the same topic, only that some people thought it was good quality and some didn't.

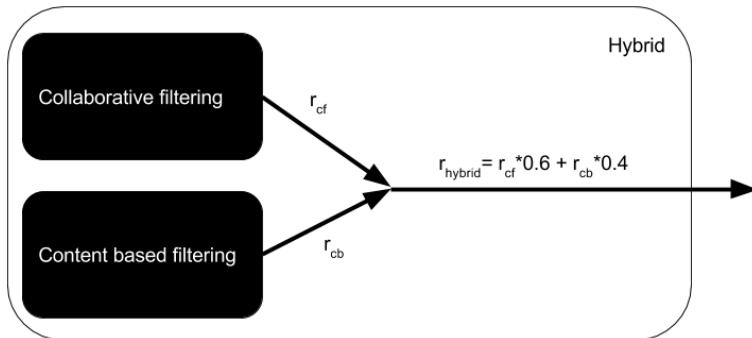


Figure 12.9 Feature weighted hybrid. Where the hybrid uses both results from a collaborative filtering and content-based filtering, using the weights 0.6 and 0.4 respectively.

We can use them together to try to combine these strengths. We don't have to give them equal power though, and this is where a weighted hybrid recommender comes into the picture. It is straightforward in the sense that you will train two different recommenders and ask them both to produce candidates for recommendations, when two or more recommenders are combined like this, then we call them feature recommenders.

Feed all of the candidates into both recommenders then take the empirical mean of the two, as illustrated in figure 12.9. To calculate an empirical mean, you select some weight like 60/40, which means that you would say predictions from this hybrid would be calculated using

$$\hat{r} = 0.6 * \overbrace{r_{\text{collaborative}}} + 0.4 * \overbrace{r_{\text{content}}}$$

The feature weights can be found in several ways, the simplest one is to guess, that is not very scientific and might result in strange recommendations. Another way is to do linear regression. And finally, you could do continuous adjustments to them using different values to different user groups and see who clicks more, using either A/B testing or multi-armed bandits (both mentioned in Chapter 9). Let's have a quick look at linear regression.

12.4.3 Linear Regression

Linear regression is about creating a function which minimizes the error between its output and the actual value. So if we have the output of two recommenders and we have the actual rating from a user, like the data in table 12.3

Table 12.73 An example of how predicted ratings from two recommenders could look compared to a users ratings.

Item	RecSys1	RecSys2	User rating
I1	4	6	5
I2	2	4	3
I3	5	5	5

Now create a function f that given output of the two recommenders predicts the user's ratings best. In other words, we want a function such that we can minimize the difference between its output and the user's rating r .

$$RSS = \sum_{\text{all ratings in training data}} (f(u, i) - r)^2$$

The function could simply look as above

$$\hat{f}(u, i) = w_1 * \text{RecSys1}(u, i) + w_2 * \text{RecSys2}(u, i)$$

So, we want to find the w such that the sum above is as small as possible. There are a lot of quite complex ways to figure out what these values should be. I think I am going to do as they do in *Introduction to Statistical Learning*⁹⁶, and move on after commenting that multiple regression coefficients estimates (which the above is) have somewhat complicated forms, and refer to the fact that there are many software packages out there that can easily solve it for you⁹⁷.

Its good having weights for each recommender, but maybe it could be even better to have this weights change depending on the user or items. This is what we will do in the following.

12.5 Feature-Weighted Linear Stacking

Above we looked at feature weighted hybrids, which means that we will combine several recommenders using a fixed weight, this is similar to doing linear functions with the output of different recommenders, for example:

$$b(u, i) = w_1 * r_{cf}(u, i) + w_2 * r_{cb}(u, i)$$

w_1, w_2 : the feature weights

⁹⁶ <http://www-bcf.usc.edu/~gareth/ISL/> great book, on machine learning.

⁹⁷ For example scikit-learn http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

$r_{cf}(u,i)$: the collaborative filtering predictor

$r_{cb}(u,i)$: the content based predictor

We call the result here, for blend, which is also a dear name for these feature weighted hybrids. In this example, we have weighted input of two recommenders (the weights are r_{cf} and r_{cb}), you can use more than two in fact there are no limits to how many you can use. The idea we are going to look at in this section is to make the weights into functions instead. It comes originally from an article by the same name by Joseph Sill et al.⁹⁸

12.5.1 Meta-features - Weights as functions

But let's say we want to make it even more flexible, for example saying that we want to use the content-based recommendation more if the user has only rated few items, and the collaborative filtering if the user has interacted with a lot of items. To do that we can extend the example above by replacing the weights with functions we would like to use, these functions are usually called meta functions or simply feature-weights functions as shown in Figure 12.10. And will look like the following:

$$b(u,i) = f(u,i) * r_{cf}(u,i) + g(u,i) * r_{cb}(u,i)$$

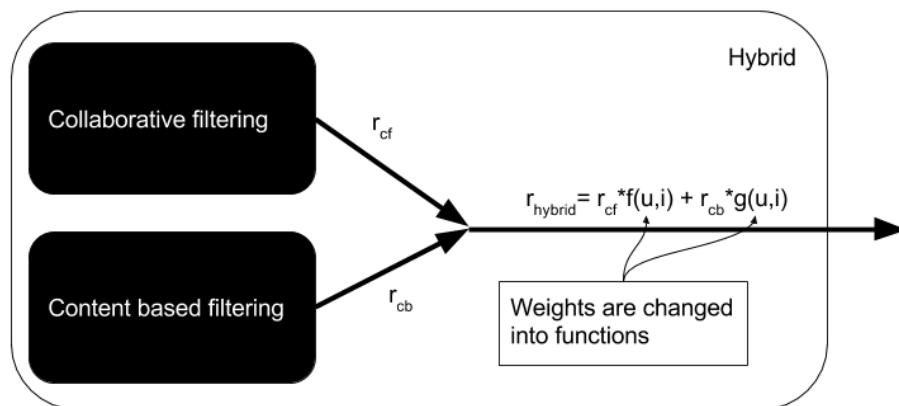


Figure 12.10 Example of a Feature-weighted linear stacking recommender.

This type of function was used to win the Netflix prize. Figure 12.11 shows a selection of meta functions used to win the Netflix Prize. They seem so simple it is hard to believe that there is

⁹⁸ <https://arxiv.org/pdf/0911.0460.pdf>

enough power in these simple functions to almost win a million dollars. Those functions deserve respect.

1	A constant 1 voting feature (this allows the original predictors to be regressed against in addition to their interaction with the voting features)
2	A binary variable indicating whether the user rated more than 3 movies on this particular date
3	The log of the number of times the movie has been rated
4	The log of the number of distinct dates on which a user has rated movies
5	A bayesian estimate of the mean rating of the movie after having subtracted out the user's bayesian-estimated mean
6	The log of the number of user ratings
7	The mean rating of the user, shrunk in a standard bayesian way towards the mean over all users of the simple averages of the users
8	The norm of the SVD factor vector of the user from a 10-factor SVD trained on the residuals of global effects
9	The norm of the SVD factor vector of the movie from a 10-factor SVD trained on the residuals of global effects
10	The log of the sum of the positive correlations of movies the user has already rated with the movie to be predicted
11	The standard deviation of the prediction of a 60-factor ordinal SVD
12	Log of the average number of user ratings for those users who rated the movie
13	The log of the standard deviation of the dates on which the movie was rated. Multiple ratings on the same date are represented multiple times in this calculation
14	The percentage of the correlation sum in feature 10 accounted for by the top 20 percent of the most correlated movies the user has rated.
15	The standard deviation of the date-specific user means from a model which has separate user means (a.k.a biases) for each date
16	The standard deviation of the user ratings
17	The standard deviation of the movie ratings
18	The log of (rating date - first user rating date + 1)
19	The log of the number of user ratings on the date + 1
20	The maximum correlation of the movie with any other movie, regardless of whether the other movies have been rated by the user or not
21	Feature 3 times Feature 6, i.e., the log of the number of user ratings times the log of the number of movie ratings
22	Among pairs of users who rated the movie, the average overlap in the sets of movies the two users rated, where overlap is defined as the percentage of movies in the smaller of the two sets which are also in the larger of the two sets.
23	The percentage of ratings of the movie which were the only rating of the day for the user
24	The (regularized) average number of movie ratings for the movies rated by the user.

Figure 12.11 Feature weights used for the Netflix prize model⁹⁹

⁹⁹ Found in the article "Feature-Weighted Linear Stacking" - <https://arxiv.org/pdf/0911.0460.pdf>

12.5.2 The algorithm

Having a function like the one shown above is nice, but how do we figure out how these functions should look? The list of functions in figure 12.11 comes from good knowledge of data but probably also a good portion of wild guessing.

We can say that we have a series of recommenders g_1, g_2, \dots, g_L . Every g_i takes an input like user and item, and returns a predicted rating. With that we can write the simple linear function like this:

$$r_{FW}(u,i) = w_1 g_1(u,i) + w_2 g_2(u,i) + \dots + w_L g_L(u,i)$$

We call it FW for feature weighting. As we have seen in earlier chapters there is an easier way to write this:

$$r_{FW}(u,i) = \sum_{j=1}^L w_j g_j(u,i)$$

That looks too simple, right? Make each of the w_j 's into a function as we saw above, which we called a feature weighting function.

$$r_{FW}(u,i) = \sum_{j=1}^L w_j(u,i) g_j(u,i)$$

Now we have a short way to write linear functions, so we can add the cool stuff.

The feature weighting functions can be defined as following. Each weight w_j is defined as a sum of functions with a weight v in front. Like the following:

$$w_j(u,i) = \sum_{k=1}^M v_{kj} f_k(u,i)$$

But this is confusing, right ?! At least I thought so the first time I saw it. Each weighting function is the sum of all the meta features. The idea is that you want to let the machine decide what is better. You can say that you want the content-based recommender to provide 90% of the answer if the user has rated less than 3 items, otherwise it should be 50/50. To solve this we could have two functions, like function 1 and 2 the following:

The first function could be:

$$f_1(u,i) = 1$$

$$f_2(u, i) = \begin{cases} 1 & \text{if } u \text{ has rated less than 3 items} \\ 0 & \text{otherwise} \end{cases}$$

We had two predictors p_{cf} – collaborative filtering and p_{cb} – content based filtering. We will can make a blended recommender function b like the following

$$b(u, i) = (v_{11} * f_1(u, i) + v_{12} * f_2(u, i)) r_{cf}(u, i) + (v_{21} * f_1(u, i) + v_{22} * f_2(u, i)) r_{cb}(u, i)$$

It's already a bit long. But can you guess what values we should give the v 's? If we set the v 's as shown in the following expression (highlighted with red), we would gain what we wanted

$$b(u, i) = (0.5 * f_1(u, i) + (-0.4) * f_2(u, i)) p_{cf}(u, i) + (0.5 * f_1(u, i) + 0.4 * f_2(u, i)) p_{cb}(u, i)$$

If the user has rated more than 3 items then $f_2(u, i) = 0$ and $f_1(u, i) = 1$:

$$b(u, i) = (0.5 * 1 + (-0.4) * 0) p_{cf}(u, i) + (0.5 * 1 + 0.4 * 0) p_{cb}(u, i) = 0.5 * p_{cf} + 0.5 * p_{cb}$$

If on the other hand the user has rated less than 3 then $f_2(u, i) = 1$ and $f_1(u, i) = 1$:

$$b(u, i) = (0.5 * 1 + (-0.4) * 1) p_{cf}(u, i) + (0.5 * 1 + 0.4 * 1) p_{cb}(u, i) = 0.1 * p_{cf} + 0.9 * p_{cb}$$

Which is what we wanted.

To get back in theory mode I will quickly take the expressions above and combine them, so we get a more compact expression of r_{FWLS}

$$\begin{aligned} r_{FW}(u, i) &= \sum_{j=1}^L w_j g_j(u, i) \\ r_{FWLS}(u, i) &= \sum_{j=1}^L \left[\sum_{k=1}^M v_{jk} f_k(u, i) \right] g_j(u, i) \\ w_j(u, i) &= \sum_{k=1}^M v_{kj} f_k(u, i) \end{aligned}$$

With this, we have Feature Weighted Linear Stacking (FWLS). Figure 12.12 illustrates this. This will allow us to blend the output of recommenders using weights which are really functions and makes it extremely flexible

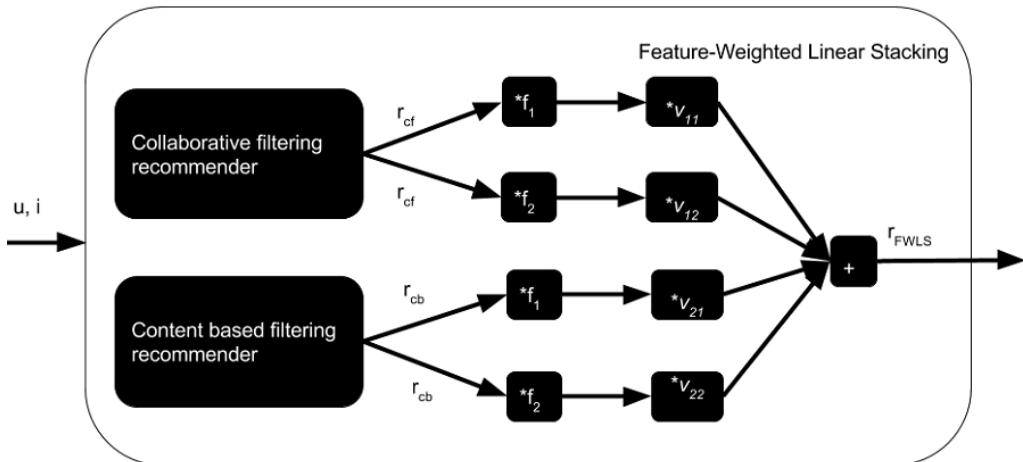


Figure 12.12 Feature Weighted Linear stacking

But manually setting the values is one thing, but we want the machine to look at the data and decide for itself which values are better. In other words, we want to train the algorithm. Let's call a shovel a shovel and just admit we are going to use machine learning.

Training the algorithm comes down to more or less the same problem as we saw in the previous chapter. We want to take the data we have in our database and use that to figure out what those weights should be. Let's take an example. We could say that we have usual user rating matrix shown in table 12.4.

Table 12.48 An example of a rating matrix



	Comedy	Action	Comedy	Action	Drama	Drama
Sara	5	3		2	2	2
Jesper	4	3	4		3	3
Therese	5	2	5	2	1	1
Helle	3	5	3		1	1
Pietro	3	3	3	2	4	5
Ekaterina	2	3	2	3	5	5

Since we have 6 users and 6 items we have $6 \times 6 - 3$ (the minus 3 is the empty cells in the table) meaning we have 33 data points, each containing a user id, item id, and a rating.

For example: (sara, star trek, 3). We consider this something true, and we can therefore say that if we input (Sara, Star Trek) into the function, then we want to get 3 out. The same with all the other 32 data points. That means that we want the hybrid recommender function should produce output that makes the difference between that expression and the actual ratings to be as small as possible. So we want to minimize the following:

$$r_{FSWL}(\text{Sara}, \text{Star Trek}) - 3$$

Well actually, we need to do something more, because if we set the rating to 0 in the expression above then the result would be -3, which is the smallest we can get. So let's aim for minimizing this instead

$$(r_{FSWL}(\text{Sara}, \text{Star Trek}) - 3)^2 \quad (1)$$

Because then we can stick to the goal of getting it as close to zero as possible.

The expression (1) above is only for one of the 33 data points we have. And we need to be sure that the function works for all of them. We want to do this for all users and all items. So we get the following

$$\sum_{u \in \text{users}} \sum_{i \in \text{items}} (r_{FSWL}(u, i) - r)^2$$

So what was the r_{FSWL} ? In the following, we have inserted it into the expression

$$\sum_{u \in \text{users}} \sum_{i \in \text{items}} \left(\sum_{j=1}^L \sum_{k=1}^M v_{jk} f_k(u, i) g_j(u, i) - r_{u,i} \right)^2$$

And this is what we want to make as small as possible. This is basically the algorithm. There are a lot of different ways to take it from here, but we will keep it simple.

$$b(u, i) = (0.5 * f_1(u, i) + (-0.4) * f_2(u, i)) p_{cf}(u, i) + (0.5 * f_1(u, i) + 0.4 * f_2(u, i)) p_{cb}(u, i)$$

If we want to know what this recommender would predict a rating for an Avengers film we would simply call both feature recommenders (assume they predicted 4 and 5 respectively), and run the two functions, one returns 1 and the other one 1 as well. That means the hybrid would calculate the following:

$$b(u, i) = (0.5 * 1 + (-0.4) * 1) * 4 + (0.5 * 1 + 0.4 * 1) * 4 = 4.9$$

Now let's go back to the weights and see how we can machine learn ourselves out of that problem.

WHEN THE FEATURE RECOMMENDER DOESN'T KNOW.

One of the things we need to think about up front is what to do when one or more of the feature recommenders doesn't respond anything.

The easiest thing to do is simply take out the data where one of the recommenders does not respond. This will reduce the training set, which might not be a viable solution. We can also try to guess by adding the average rating of the user, the item, the average of the two, or the baseline predictors described in chapter 11.

In the following implementation, we will take the first solution and simply remove all the rows that don't contain predicted ratings from either of the feature recommenders.

It could also be that a feature weighting function is not defined for all rows; in that case, you can also either remove the row or come up with some default value.

TRAINING THE BEAST

To prepare the hybrid recommender, we need to do a few things. It can be confusing and easy to make small mistakes which become big ones when you have a long pipeline like this one.

Figure 12.12 illustrates the pipeline. We will go through the following steps:

- Train the featured recommenders
- Generate predictions for each of the training data points
- Execute each feature-weighted function on all training, data points
- For each training data-point calculate the product between each predicted rating with each feature weighted function result
- Split data in train and test data (not shown in figure 12.13)
- Find the unknown values using linear regression
- Test the hybrid on the test set

There is one question that needs to be answered before we can go on. What do we do in cases where the recommender does not return an answer?

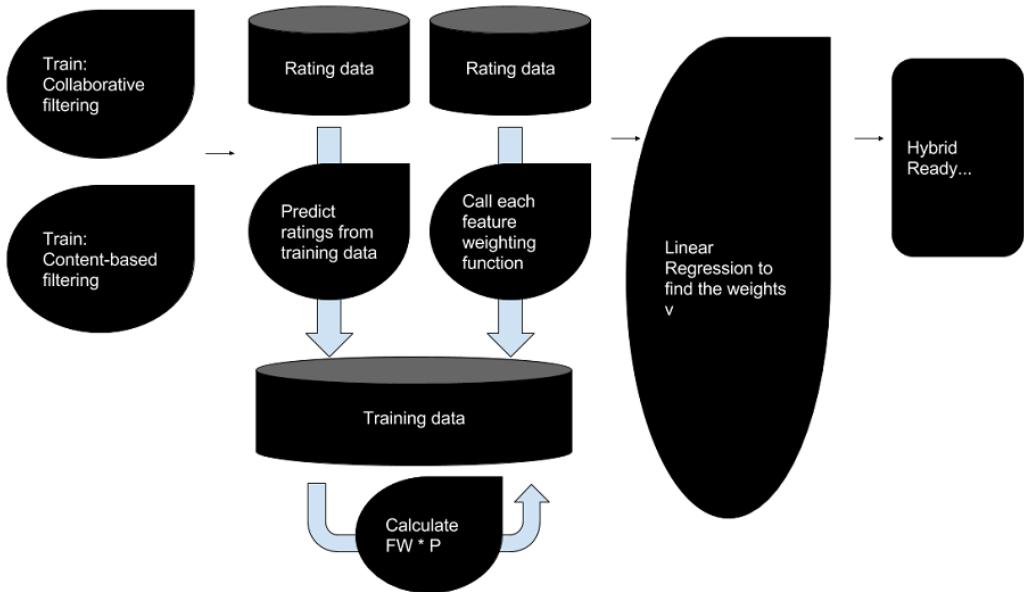


Figure 12.13 The Feature Weighted Linear Stacking hybrid recommender training pipeline.

TRAIN THE FEATURED RECOMMENDERS

Here we are going to need a lot of the things we learned in earlier chapters. Before we can even begin on the hybrid recommender stuff we need to prepare the recommenders that the hybrid will use. First we will prepare the item-item collaborative filtering recommender (which was described in chapter 8), and the content-based recommender (chapter 10). We could also have used something else, but those were the two I choose.

GENERATE PREDICTIONS FOR EACH OF THE TRAINING DATA POINTS

Here we could arrive at a strange problem: how will your recommender react when you try to predict ratings that already exist? Depending on how you have implemented it, your collaborative filtering system might decide to use the most similar item it can find to the current item, and that is the same, and the recommender will therefore return the actual rating instead of a prediction. That is not a good idea, since it will skew the training of the hybrid. We will pretend that the current item has not been rated by the current user.

So we have the ratings in the following format:

Table 12.1 Training data for Men in Black

user	item	Actual rating
Sara	MIB	5
Jesper	MIB	4
Therese	MIB	5
Helle	MIB	4

When the recommenders are trained, and they are ready to shine, we will take the training data, which is the data we will use to teach the hybrid algorithm to recommend, and run it by each of the recommenders, and see what they would predict for each of user-item pairs. This will give us the following table:

Table 12.2 Training data with the predictions added

user	item	Prediction		Actual rating
		Collaborative Filtering	Prediction Content based	
Sara	MIB	4.5	3.5	5
Jesper	MIB	4	5	4
Therese	MIB	4	4	5
Helle	MIB	3	5	4

EXECUTE EACH FEATURE-WEIGHTED FUNCTION ON ALL TRAINING DATA POINTS

And then we want to run it past the two functions to get their results also.

Table 12.3 Training data with the predictions and function results added

user	item	Prediction		F1	F2	Actual rating
		Collaborative Filtering	Prediction cb			
Sara	MIB	4.5	3.5	1	0	5
Jesper	MIB	4	5	1	1	4
Therese	MIB	4	4	1	1	5
Helle	MIB	4	3	1	1	3

FOR EACH TRAINING DATA POINT CALCULATE THE PRODUCT BETWEEN EACH PREDICTED RATING WITH EACH FEATURE WEIGHTED FUNCTION RESULT

Then multiply the predictions with the meta feature functions.

SPLIT DATA.

Now that data is ready, it's a good idea to split the data, and take away something like 20% of the data to use to check how well the algorithm does. The problem is that if we use all of the data to train the algorithm, then we will measure its power on data that is already seen. Instead, we need some data left out, so we can see how good it will do on not seen data.

FIND THE UNKNOWN VALUES V'S USING LINEAR REGRESSION.

To find the v 's we use linear regression. Linear regression means that we will try to find a line which is as close to the actual data as possible. Such a line might not be possible to find, but we will get back to that at a later time.

Linear regression is one of the first things that you will learn in any machine learning course. And there is written whole books about the subject¹⁰⁰, so I won't try to teach it here. Other than just tell you that the idea is basically the same as we talked about in the previous chapter, when we tried to find the unknowns in the FunkSVD. In the sense that it will try to minimize the squared error over all data points.

TEST THE HYBRID ON THE TEST SET.

When the linear regression is finished, then we can check the quality (compared to the actual ratings of the test set (the data we left out)) and decide whether it works for us. We will check the quality of the hybrid by checking how close the predictions of the hybrid fits the actual ratings.

Now that you know it all, it's time to look at the code that could be used to implement it.

12.6 Implementation

Let's see how a Feature Weighted Linear Stacked hybrid runs in the MovieGEEKs. We will follow the same steps as above, just describing code instead. If you haven't downloaded MovieGEEKs code yet, I suggest doing it now, so you can follow what happens. MovieGEEKs is implemented to be ready and running in just a few steps. You can find it here:

<https://github.com/practical-recommender-systems/moviegeek>

When you have downloaded the code, please follow the install instructions in the readme file. During the installation you will download the MovieTweetings dataset.

TRAIN THE FEATURED RECOMMENDERS

Train the item-item collaborative filtering by running the Item Similarity Calculator

¹⁰⁰ *Introduction to statistical learning* by G. James et al, is not only about linear regression but it has a good chapter about it.

Run

```
python -m builder.item_similarity_calculator
```

And build an LDA model for use the content based filtering.

Run

```
python -m builder.lda_model_builder
```

We will now load some rating data, which I want to be sure you understand is made up of the ratings collected from the users (either explicit or implicit). This is used to find the tune the hybrid, by providing both input and expected output.

Listing 12.1: builder/fwls_calculator.py - Load data

```
def get_real_training_data(self):
    columns = ['user_id', 'movie_id', 'rating', 'type']
    ratings_data = Rating.objects.all().values(*columns)
    df = pd.DataFrame.from_records(ratings_data, columns=columns) ①
```

① Create a panda Dataframe of the data.

To get the data, we loaded all the ratings into memory. If there are red flags waving here, then there should be. It is not a good idea to load the whole rating table into memory unless you are asking for trouble. But the size we have here is okay, if not then you will have to try to either stream the data, chunk by chunk, otherwise you can take a sample of the data. Maybe the simplest thing would be to cut away old data, until you can fit it in memory.

Moving on.

GENERATE PREDICTIONS FOR EACH OF THE TRAINING DATA POINTS.

With the ratings loaded we can take each of the data points in the training data, and see what each of the feature recommenders would predict. These will be added to the data frame.

Here are the methods which are called to do the predictions.

The first one is the collaborative filtering, which we described in lots of detail in chapter 8, so please refer to that chapter for more info. The code was the following:

Listing 12.2: recs/neighbourhood_based_recommender.py

```
def predict_score_by_ratings(self, item_id, movie_ids):
    top = 0
    bottom = 0

    candidate_items =
        Similarity.objects.filter(source__in=movie_ids.keys()).filter(target=item_id)
    candidate_items = candidate_items.distinct().order_by('-
        similarity')[self.max_candidates]

    if len(candidate_items) == 0:
```

```

    return 0

    for sim_item in candidate_items:
        r = movie_ids[sim_item.source]
        top += sim_item.similarity * r
        bottom += sim_item.similarity

    return top/bottom

```

The second recommender is the content based recommender that uses the LDA, this was described in lots of detail in chapter 10, but not actually this method. What happens is that you use the similarities and the user's ratings to predict a rating, here you just use the similarity from the LDA instead of the similarity based on user behaviour as in collaborative filtering.

Listing 12.2: recs/content based_recommender.py

```

def predict_score(self, user_id, item_id):

    active_user_items = Rating.objects.filter(user_id=user_id).order_by('-rating')[:100]

    movie_ids = {movie['movie_id']: movie['rating'] for movie in active_user_items}
    user_mean = sum(movie_ids.values()) / len(movie_ids)

    sims = LdaSimilarity.objects.filter(Q(source__in=movie_ids.keys())
                                         & Q(target=item_id)
                                         & Q(similarity__gt=self.min_sim)
                                         ).order_by('-similarity')

    pre = 0
    sim_sum = 0

    if len(sims) > 0:
        for sim_item in sims:
            r = Decimal(movie_ids[sim_item.source] - user_mean)
            pre += sim_item.similarity * r
            sim_sum += sim_item.similarity

    return Decimal(user_mean) + pre / sim_sum

```

We will use these two methods on each of the ratings in the training data. Simply by applying it to each row, and inserting the answer into new columns

Listing 12.3: builder/fwls_calculator.py – calculate_predictions_for_training_data

```

def calculate_predictions_for_training_data(self):

    self.training_data['cb'] = self.training_data.apply(lambda data:
self.cb.predict_score(data['user_id'], data['movie_id']))
    self.training_data['cf'] = self.training_data.apply(lambda data:
self.cf.predict_score(data['user_id'], data['movie_id']))

```

We now have structure like the one we saw in Table 12.2

EXECUTE EACH FEATURE WEIGHTED FUNCTION ON ALL TRAINING DATA POINTS

The functions we will use are very simple and not really much to talk about. But I deem that they are enough to show how feature-weighted linear stacking works.

Listing 12.4: builder/fwls_calculator.py – functions

```
def fun1(self):
    return 1.0

def fun2(self, user_id):
    if self.rating_count[self.rating_count['user_id']==user_id]['movie_id'].values[0] < 3.0:
        return 1.0
    return 0.0
```

Instead of calculating the function first and save that data we will use the functions and do the product of their result in one step.

FOR EACH TRAINING DATA POINT, CALCULATE THE PRODUCT BETWEEN EACH PREDICTED RATING WITH EACH FUNCTION

So, for each prediction we want to have column with the product of the prediction and each of the functions. We have two predictors and two functions, meaning we need to have 4 new columns.

Listing 12.5: builder/fwls_calculator.py – functions

```
def calculate_feature_functions_for_training_data(self):
    self.training_data['cb1'] = self.training_data.apply(lambda data:
                                                          data.cb*self.func1()) ①
    self.training_data['cb2'] = self.training_data.apply(lambda data:
                                                          data.cb*self.func2(data['user_id'])) ②

    self.training_data['cf1'] = self.training_data.apply(lambda data:
                                                          data.cf*self.func1()) ③
    self.training_data['cf2'] = self.training_data.apply(lambda data:
                                                          data.cf*self.func2(data['user_id'])) ④
```

- ① calculating the product of the content based prediction and function 1.
- ② calculating the product of the content based prediction and function 2.
- ③ calculating the product of the neighbourhood based prediction and function 1.
- ④ calculating the product of the neighbourhood based prediction and function 2.

We are finished with preparing for doing the linear regression to find the v's. This is also called feature generation. Often when you do machine learning applications this is where you will spend a lot of time.

FEATURE WEIGHTED FUNCTION RESULT

We cheated and did that in the step above. It might not always be a good idea to do this in real life. Sometimes it is better to do only small steps. It depends on your specific scenario.

SPLIT DATA

We split the data into train and test data. This is done to enable us to see how good the result of our work is. We use the training data to select some good values for the weights and then the test set to see how good we did.

Listing 12.6: builder/fwls_calculator.py – Split data

```
self.train, self.test = train_test_split(self.train, test_size = 0.2) ①
```

- ① split the data into train and test data.

FIND THE V'S USING LINEAR REGRESSION.

We now have data which can be used to find the elusive weights.

Listing 12.7: builder/fwls_calculator.py– regression

```
def train(self):
    regr = linear_model.LinearRegression(fit_intercept=True,
                                         n_jobs=-1,
                                         normalize=True)
    regr.fit(self.train_data[['cb1', 'cb2', 'cf1', 'cf2']],
             self.train_data['rating']) ②
```

- ① Instantiate the LinearRegression Model class of scikit learn¹⁰¹.
- ② fit the weights to build the linear function.

To use these weights, we need to save them, so let's put them in the database so that we can use them again at a later time.

THE ONLINE RECOMMENDATION PREDICTION

The hybrid recommender, could be done by pre-calculating recommendations and saving them in the database. The lazier approach is to call all the feature recommenders and then mix the results.

To be sure the ordering will be good, both recommenders is requested to provide a top N which is five times larger than what the user is requesting. This is to allow for the linear functions enough elements to order them correctly.

Listing 12.8: recs/fwls_recommender.py

```
def recommend_items(self, user_id, num=6):
    cb_recs = self.cb.recommend_items(user_id, num * 5) ①
    cf_recs = self.cf.recommend_items(user_id, num * 5) ②
```

¹⁰¹ http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

```

combined_recs = dict()
for rec in cb_recs:
    movie_id = rec[0]
    pred = rec[1]['prediction']
    combined_recs[movie_id] = {'cb': pred} ③

for rec in cf_recs: :
    movie_id = rec[0]
    pred = rec[1]['prediction']
    if movie_id in combined_recs.keys():
        combined_recs[movie_id]['cf'] = pred
    else:
        combined_recs[movie_id] = {'cf': pred}
fwls_preds = dict()
for key, recs in combined_recs.items():
    if 'cb' not in recs.keys():
        recs['cb'] = self.cb.predict_score(user_id, key)
    if 'cf' not in recs.keys():
        recs['cf'] = self.cf.predict_score(user_id, key)
    pred = self.prediction(recs['cb'], recs['cf'], user_id) ⑥
    fwls_preds[key] = {'prediction': pred}
sorted_items = sorted(fwls_preds.items(),
                      key=lambda item: -float(item[1]['prediction']))[:num] ⑦

return sorted_items

```

- ① Call the content based recommender requesting 5 times more elements than needed.
- ② Call the neighbourhood based recommender requesting 5 times more elements than needed.
- ③ now run through all the recommendations and create a dictionary of dictionaries. Start out adding all items from the content based recommendation.
- ④ now add all the items from the neighbourhood model.
- ⑤ run through the model and get rating predictions from the recommenders that is missing.
- ⑥ Now calculate the predicted rating for all elements in the dictionary, using the predict method, shown below.
- ⑦ order result by prediction.

Listing 12.9: recs/fwls_recommender.py

```

def prediction(self, p_cb, p_cf, user_id):
    p = (self.wcb1 * self.fun1() * p_cb +
         self.wcb2 * self.fun2(user_id) * p_cb +
         self.wcf1 * self.fun1() * p_cf +
         self.wcf2 * self.fun2(user_id) * p_cf) ①
    return p + self.intercept

def fun1(self):
    return Decimal(1.0) ②

def fun2(self, user_id):
    count = Rating.objects.filter(user_id=user_id).count()
    if count > 3.0:
        return Decimal(1.0) ③
    return Decimal(0.0)

```

- ① calculate the predicted rating for the item.
- ② function 1

③ function 2

In the following you will see a few recommendations produced with the hybrid.

HOW DOES THE HYBRID COMPARE

One thing is that they are correlated, but how does the hybrid compare with the individual recommenders? To understand that we could again see which of the 3 (collaborative, content-based and hybrid) come closer to the actual ratings.

First the example where the user has only one rating, then the neighborhood model doesn't return anything. An example of such is shown in figure



Figure 12.14 Example of recs done by the hybrid recommender

If on the other hand we look at a user which has several ratings registered, then the combination of the two recommenders end up showing items which would not have entered into the top 6 of the content-based nor the neighborhood-based recommenders. As can be seen in figure 12.16

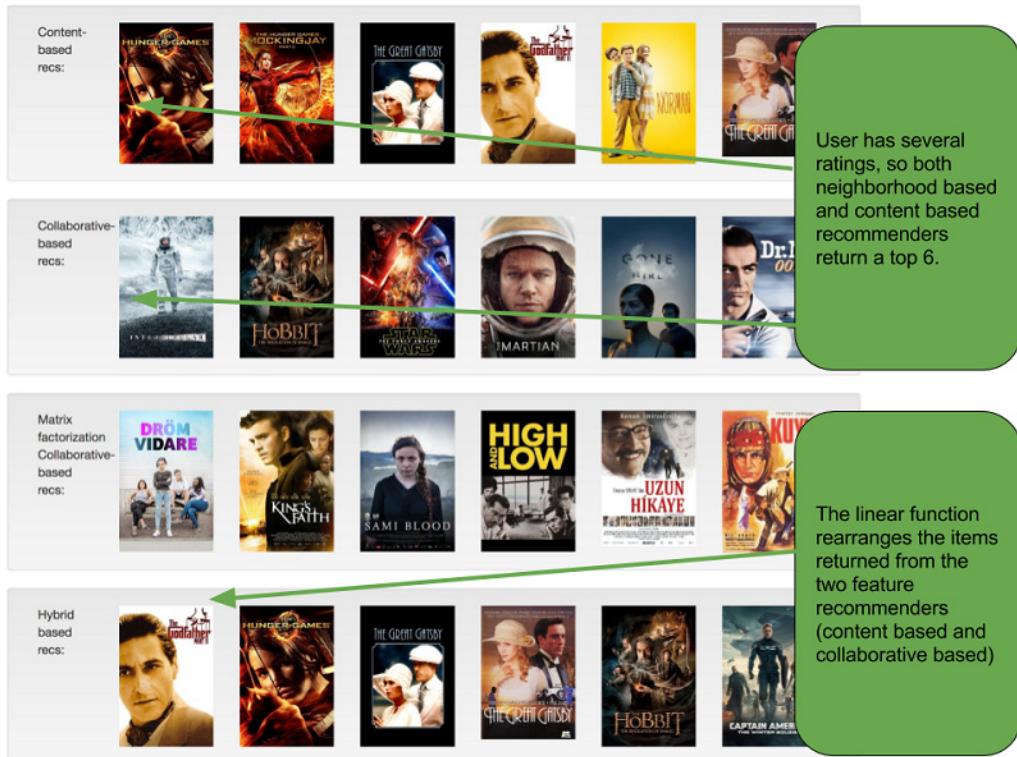


Figure 12.15 Showing the recommendations from the hybrid recommender, where the both feature recommenders have predicted recs.

TEST THE HYBRID ON THE TEST SET.

So, did it work...? How do we test that? Well there is the offline evaluation which you learned about in chapter 9. Where you do cross validation. As shown in figure 12.13

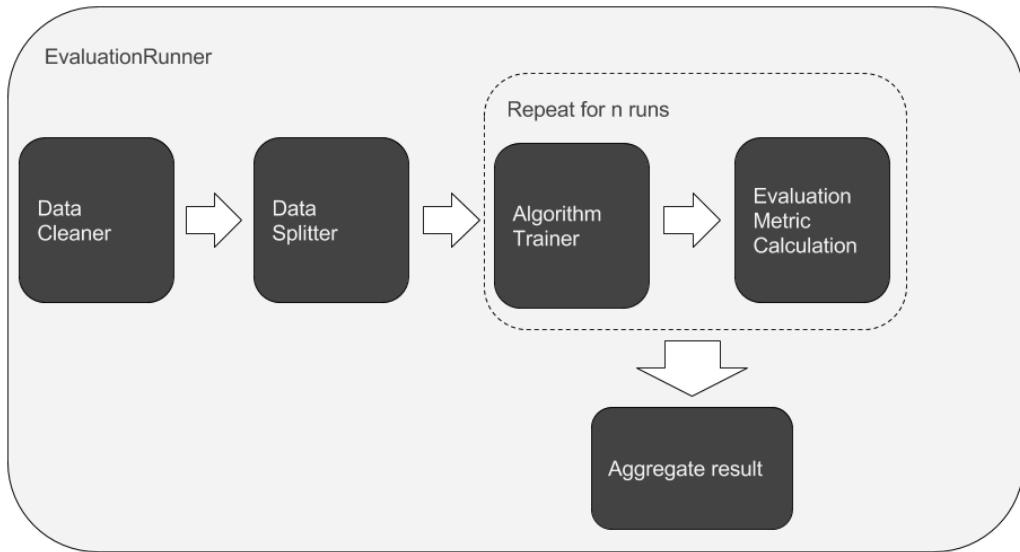


Figure 12.16 The evaluation runner is used to evaluate an algorithm. It is a pipeline where the data is first cleaned, then split into the k folds for cross validation. Then for each fold it repeats training of the algorithm, evaluating the algorithm. And then finally when it is all finished, you aggregate the result.

Running this evaluation runner is almost like described in chapter 9, only here you need to train all of the included recommenders.

You can run the evaluation by executing the following:

Listing 12.13: executing the evaluation.

```
> python -m evaluator.evaluation_runner -fwls
```

It will produce data as shown in figure 12.18

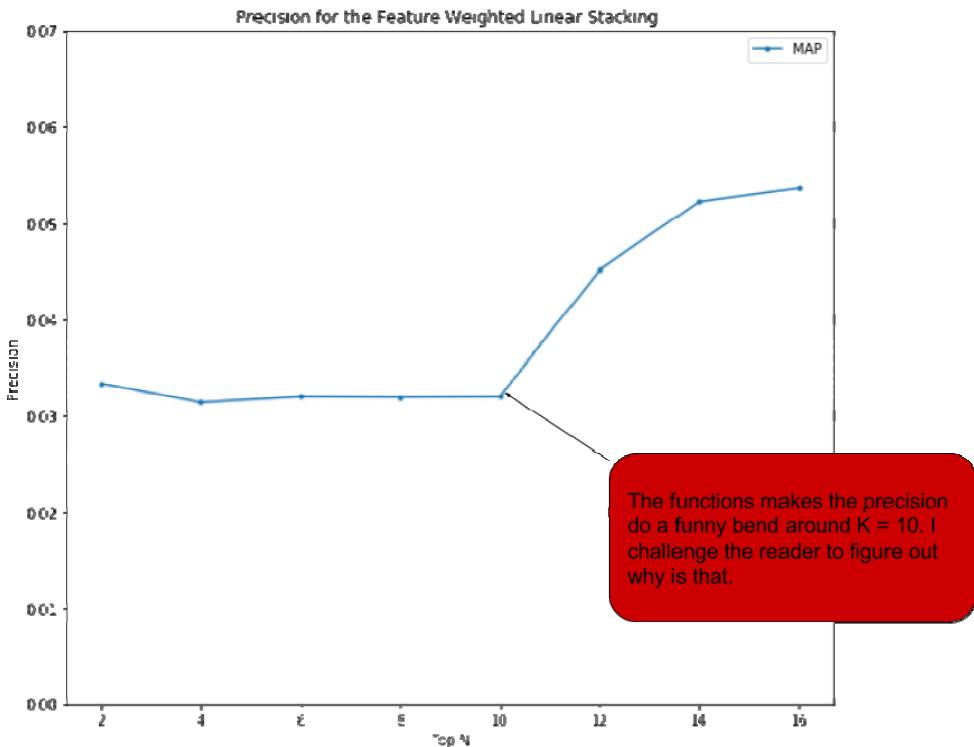


Figure 12.17 The evaluation of the FWLS algorithm.

The evaluation that is shown above is a bit special compared to the others we have looked at in the book. Because the two recommender systems are build, and then we do linear regression to find the weights for a function which produces the final recommender system. If I were to put a system like this in production, then I would probably pre-calculate a lot of this to speed things up. A way to make the training a bit faster here, is to just sample the ratings used to do the linear regression. Like I have done here.

Listing 12.14: executing the evaluation.

```
self.train_data = self.train_data.sample(self.data_size)
```

The question is how big the size should be of the sample to be representative of the who dataset. Considering that it is just the four weights you are training. It is a typical dilemma to be in with machine learning. And I can say that I had good experiences with a few hundred

data points. Which is not a lot. But if you look at the precision in the chart above, then it looks pretty good.

12.7 Summary

Now, this was a lot of different terms and combinations that were thrown around in this chapter. If you are new to linear regression, then I highly recommend looking into it. It's quite simple but can be used for so many things, that it's simply silly not to have it in your tool box.

After finishing this you should have the following:

- A recommender system can be greatly optimized by adding the output of several algorithms.
- Hybrid recommenders will enable you to combine the forces of different recommenders to get better results.
- Not all things complex is actually functional in the real world, while the algorithm that won the Netflix prize, will won the prize, it failed as it was too complicated to be put in production.
- Feature Weighted Linear stacking algorithm, enables the system to use the feature recommenders in a function weighted way, which makes it very strong.

13

Ranking and Learning to Rank

This book is all about learning, and in this chapter we will look at how to learn how to rank:

- You will reformulate the recommender problem to a ranking problem.
- You will look at the Foursquare's ranking method, and how they use multiple sources.
- You'll go through the different types of learning-to-rank algorithms, and how to distinguish pointwise, pairwise, and Listwise comparisons of ranks.
- Bayesian Personalized Ranking algorithm is a very promising algorithm, which you will go into great detail with, both in theory and in an implementation.

13.1 Introduction

Are all of these chapters on recommender algorithms starting to look the same? If so, you are in luck because now we are going to start on something completely different.

Instead of focusing on recommendations as a rating prediction problem, it sometimes makes more sense to look at how the items should be stacked. The catalog item that the user would find most relevant right now is on the top, the second one next, and so on. To define it like this takes away any need for rating prediction; we don't need to know how favorably the user would rate something, only that she would love it, or at least like it more than everything else that's available.

One should always keep in mind that the catalog of content might not contain anything the user would love, but even when that is the case we still want to provide a list of the best we can do with what we got.

I think this is going to be a very exciting chapter; we are going to talk about a type of algorithm first introduced in the area of information retrieval systems(IR) - a posh word for

search engines these days. Ranking is powering the Microsoft search engine Bing as well as most other search engines, Facebook and Foursquare use it, too. There is, of course, a difference between what they want to find and what a recommender wants, but in the end, a lot of the research done in the IR world has also been usable for recommenders. We will start this journey with an example of learning to rank from Foursquare to give you a sense of ranking. We will then put it in relation to other Learning to Rank algorithms and describe the three different overall types of Learning to Rank algorithms. To have a concrete example of a "learning to rank" algorithm, we will go into detail with the Bayesian Personalized Ranking algorithm; it has a bit of complicated math, but we will refresh it with coding the algorithm in MovieGEEK's to enable you to see it in action.

13.2 Learning to Rank example at Foursquare

Foursquare is a guide to cities. I use it to find places where I can maintain my coffee addiction (which I promise myself to stop when I finish writing this book). Imagine I am standing in front of the beautiful St. Peter's Basilica in Rome. After an exhausting, long wait in a queue to see the inside of the church I decide I need some coffee, so I flip out my phone and open the Foursquare app and click coffee near me. The result is shown in figure 13.1 (not completely but it's the browser version of Foursquare's search for the same thing).

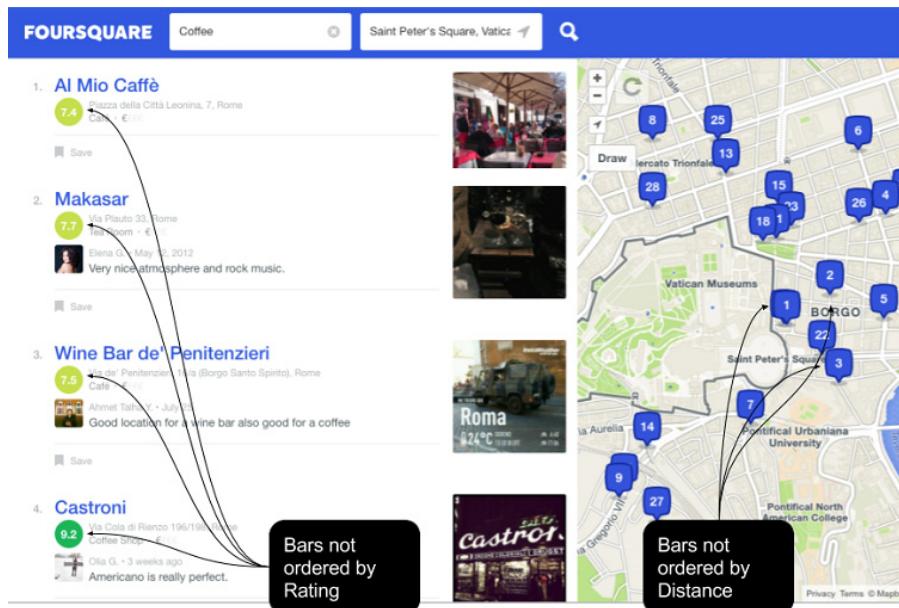


Figure 13.1 Looking for coffee place near Saint Peter's Basilica in Rome using Foursquare

As you can see, the recommendations are not ordered by rating or by distance, so how are they ordered? How did Foursquare come up with this list?

Foursquare did actually publish an excellent article on how their recommender system works. Where they describe their implementation of exactly "Learning to Rank"¹⁰². I recommend that you read it; it is a fascinating insight into the challenges of finding points of interest near users. We will look at a slightly simpler version than what they use, to enable us to give a simple introductory example of Learning to Rank.

Like the hybrid recommenders we talked about in the previous chapter, learning to rank is a way to combine different kinds of data sources, like popularity, distance, or recommender system outputs. The difference here is that it doesn't necessarily have to be from (or parts of) a recommender system. In the case of learning to rank, we are simply looking for input sources that will give us some ordering on the objects. Figure 13.2 shows an overview of the list of features (which is called features in machine learning lingo) which according to the article are utilized at Foursquare

Feature	Description
Spatial score	$P(l v)$
Timeliness	$P(t v)$
Popularity	Smoothed estimate of expected check-ins/day at the venue
Here now	# of users currently checked in to the venue at query time
Personal History	# of previous visits from the user at the venue
Creator	1 if the user created the venue, 0 otherwise
Mayor	1 is the user is the mayor of the venue, 0 otherwise ⁵
Friends Here Now	# of the user's friends currently checked in at query time
Personal History w/ Time of Day	# of previous visits from the user at the venue at the same time of the day

Figure 13.2 List of features used in Foursquare's algorithm to rank venues near you

We don't have access to the features listed in Figure 13.2 so let's try to stick to two features and see if we can make sense of the ranking of the cafes I got on the page shown in Figure 13.1.

The page shows the average rating of each venue; I found the walking distance using Google maps. If we put those data into a table it would look like the following:

Ranking on Foursquare		walking time (distance)	Average ratings
1	Al Mio Caffe	2	7.4
2	Makasar	4	7.7
3	Wine Bar de' Penitenzieri	4	7.5
4	Castroni	10	9.2

¹⁰² Shaw et al. "Learning to Rank for Spatiotemporal Search "http://courses.cse.tamu.edu/caverlee/csce670_2013/ltr-foursquare-paper.pdf

If we look at the table, it's easy to see that the ranking is not based on ratings. If it were, Castroni would be at the top. It's also not ranked by distance, or Makasar would have to share his second place with the wine bar. Let's try to do some feature engineering here and see if we can get closer to predicting the ranking for the four elements using only these two features.

First, we need both to massage the data so that a higher value denotes shorter distance and higher average rating. Being far away is not a good thing for a café, so we need to invert the distances. Inverting the distance will make places with a short walking time have a high value. To do this, find the maximum, which is a 10 min walk, and subtract each walking time from maximum. This will make the distance value of "Al mio Caffe" be $10 - 2 = 8$ and Castroni $10 - 10 = 0$. We rescale¹⁰³ all the data so that everything is between 1 and 0. Because if we don't then some algorithms might not work well. Rescaling can be done using the following formula:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Normalizing the data will give us the following:

Ranking on Foursquare		walking time (distance)	Average ratings
1	Al Mio Caffe	1.00	0.00
2	Makasar	0.75	0.17
3	Wine Bar de' Penitenzieri	0.75	0.06
4	Castroni	0.00	1.00

With this change then we have a distance ordering very close to the ranking on Foursquare. Items 2 and 3 are tied for walking time, so we need to get that from the ratings instead.

We are now at the core of the problem; we want to teach the machine to rank these items based on the input of ratings and distance.

We can formalize it a bit more by saying we want the system to learn weights (w_0 and w_1), which inserted into the following expression will produce a value such that the four items get ordered like the Foursquare page:

$$f(distance, rating) = w_0 * distance + w_1 * rating$$

We want to make the function produce an ordering like Foursquare's, so we are trying to make an algorithm that is optimized to rank based on their output. In our example, it's not too hard

¹⁰³ https://en.wikipedia.org/wiki/Feature_scaling

to guess and if you set them equal to and you will get the score values shown in the table below:

Ranking on Foursquare		walking time (distance)	Average ratings	Score
1	Al Mio Caffe	1	0	20
2	Makasar	0.75	0.17	16.7
3	Wine Bar de' Penitenzieri	0.75	0.06	15.6
4	Castroni	0	1	10

Another way to approach the problem is to use linear regression find the line that best represents the data set. Use the line to find the rank of each item by starting at the point furthest away from (0,0) and working inward. This is shown in figure 13.3. The angle of line decides which of the two features (ratings or distance) will be given most importance.

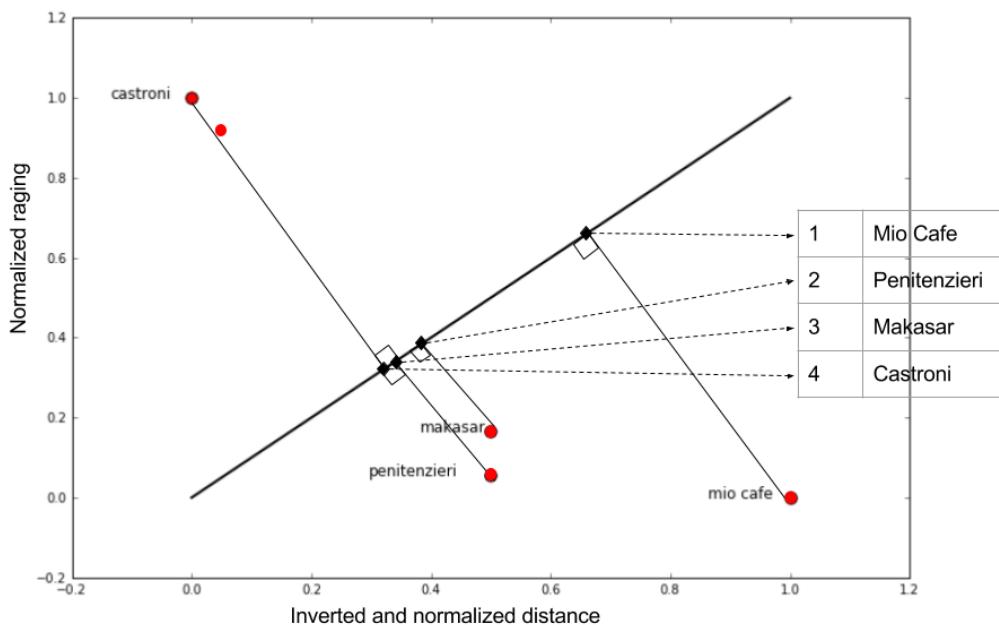


Figure 13.3 Projecting the points to a line, gives an ordering of the items.

Looking at figure 13.3 also provides us with a view of what we are trying to solve here. We have two different dimensions, the distance and the average ratings. By drawing the line as I did I got the ordering as shown in the figure, if you change the angle of the line the cafes might come out in a different order. Hope that helped.

Back to the Foursquare example: If we were Foursquare, we would probably have pipeline like the following:

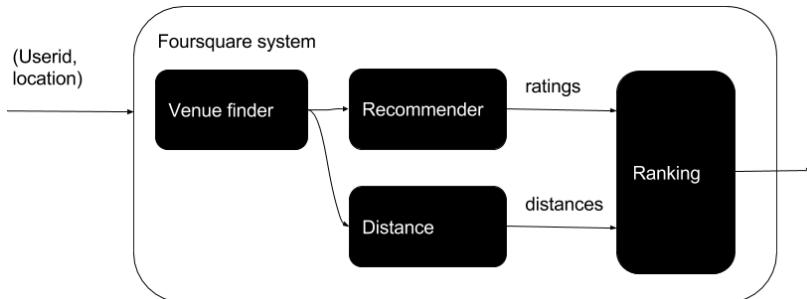


Figure 13.284 A simplified view of the Foursquare ranking system

Now the problem becomes slightly different because how do we (pretending we are Foursquare) optimize the function if we don't know what it is supposed to return? We use the check-in feature, read the articles for more details, here I just want you to note that finding data that describes how it should look are not always straight forward.

13.3 Re-ranking

If you just read the previous chapter on hybrids you might ask what the difference between this and the feature weighted hybrids is. Remember that we are optimizing for two different things. A hybrid recommender is always predicting ratings, while a Learning to Rank algorithm produces orderings. We will look at how you define ordering in the next section.

Some would call the Foursquare ranking a Re-ranking, because it is taking list of venues that is found using a spatial index, meaning that it finds the items closer to you, and then reordering the list to also match the rating criteria.

A simple example of re-ranking in the recommender system setting would be to use a popularity ordering as the base and then re-rank the items using a recommender system. The popularity will narrow the list to outputting the most popular items and reduce the risk for showing items that are very particular (and maybe unpopular) tastes. This might give you a bad smell of a filter bubble, but remember that if a user likes unusual items more than popular items, the unusual ones would still bubble up in the list. As an example, have a look at figure 13.3 again, and find *Castroni*. It is very far away, but since its average rating is so high it manages to get on the top 4 list. On the other hand, the *Mio Café* doesn't have very good ratings, but we were basically standing in the café, so even if it is unpopular, it came first because it was the closest option.

Collaborative filtering algorithms are prone to recommend content liked by very few people but people who like that content a lot. The algorithm has no concept of popularity and could,

therefore, benefit from being used for re-ranking instead of the sole source for the ordering. This example is also described on the Netflix tech blog¹⁰⁴ -

Instead of re-ranking items, why not start out with the aim of ranking instead, and then construct algorithms that optimize for that instead of rating prediction? This is the goal of "learning to rank" algorithms.

13.4 What is learning to rank?

A recommender or another type of data-driven application that produces ranked lists is trained using a family of algorithms called "Learning to Rank."

A ranking recommender system has a catalog of items. Given a user, the system will retrieve items that are relevant to the user and then rank them so the items at the top are the most relevant. The ranking is done using a ranking model¹⁰⁵. A ranking model is trained using a Learning to Rank algorithm, which is a supervised learning algorithm, meaning that we provide it with data containing input and output. In our case that is a user_id as input and a ranked list of items as output.

This family of algorithms has three subgroups, which will quickly be reviewed here.

13.4.1 The three types of learning to Rank algorithms

These algorithms are distinguished by the way they evaluate the ranked list during training. Figure 13.5 illustrates the three different flavors.

¹⁰⁴ <http://techblog.netflix.com/2012/06/netflix-recommendations-beyond-5-stars.html>

¹⁰⁵ This definition is loosely taken from the article "A Short Introduction to Learning to Rank" <http://times.cs.uiuc.edu/course/598f14/l2r.pdf>

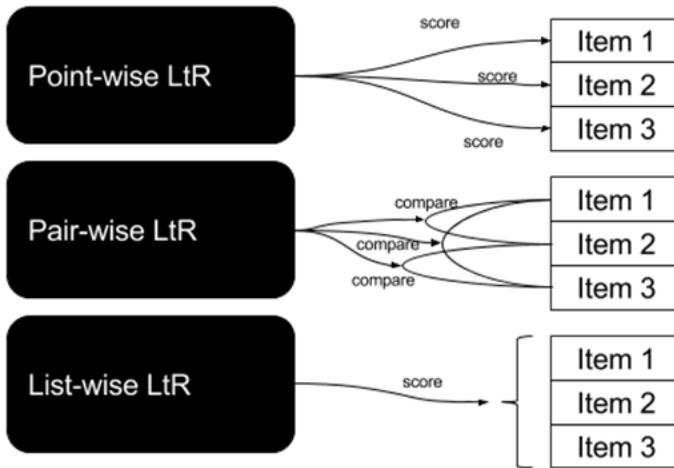


Figure 13.5 The three different subgroups of learning to rank – pointwise, pairwise, and listwise

POINTWISE

This approach is the same as the recommenders we have looked at in earlier chapters; it produces a score for each item and then ranks them accordingly. The difference between rating prediction and ranking is that with ranking, we don't care if an item has a utility score of a million, or within some rating scale, as long as the score symbolizes some position in the rank.

PAIRWISE

This is a type of binary classifier; it's a function that will take two items and return an ordering of the two.

When you talk about pairwise ranking, you usually optimize the output in order to have the minimal number of inversions of items compared to the optimal rank of the items. An inversion means that two items change places.

To do pairwise ordering, we need what is called an absolute ordering. An absolute ordering means that for any two content items in the catalog, we can say one is more relevant than the other or tied. If we made a pairwise ranking simply by predicting ratings using the neighborhood model from chapter 10 we would not have an absolute ordering since the algorithm can't predict ratings for all items.

LISTWISE

Finally, listwise is the king of them all, as it looks at the whole ranked list and optimizes that. The advantage of listwise ranking is that it includes the fact that ordering is more important at the top of a ranked list compared to the bottom. The pointwise and pairwise algorithms don't

distinguish where on the ranked list you are. Consider, for example, a top-10 recommendation, the pairwise recommendation will punish you the same for getting the order of the last two items wrong as much as getting the two first one's wrong. We know by now that that is not good, as users will pay a lot more attention to top. To inject this into the algorithm also, means that we need to look at the complete list and not each pair of items. It sounds quite simple, when you explain it like this. You just have to create a ranking such that all items are always ranked correctly, but it turns out that it is quite hard to programmatically calculate whether one list is better than another one. To have a look at a listwise ranking algorithm, I can suggest cofiRank, which was presented at NIPS 2007¹⁰⁶

In the following we will look more in detail at an algorithm which uses pairwise ranking.

13.5 Bayesian Personalized Ranking

Once again, it's always a good idea to be sure we agree on the problem. That is true for this and so many other things in life. Let's start out with a definition of the problem or task we want to solve, we will use an algorithm called Bayesian Personalized Ranking (BPR), which was presented in a paper by Steffen Rendle et al.¹⁰⁷

TASK TO SOLVE

The overall idea is that we want to provide customers with a list of items where the top one is the most relevant one, then the next best one and so on. So far I am pretty sure we understood each other. So for each user we want to be able to order the content items in such a way that the most relevant is on top. This we need to describe in a way that both you and the machine will understand.

We need to define an ordering, which will tell us no matter which two items we hold up, then the ordering will say one is better than the other. To make that work, we need three rules to be upheld: Totality, Anti-symmetry, and Transitivity.

For a given user we want an ordering written like $>_u$ is defined as following:

- **Totality:** For all i, j in I we have that if $i \neq j$ then we have either $i >_u j$ or $j >_u i$.
- **Anti-symmetry:** For all i, j in I we have that if $i >_u j$ and $j >_u i$ then $i = j$
- **Transitivity:** For all i, j, k in I we have if $i >_u j$ and $j >_u k$ then $i >_u k$

This might not seem like something very important to spend so much time on, but it is needed to make it work.

¹⁰⁶ <http://papers.nips.cc/paper/3359-cofi-rank-maximum-margin-matrix-factorization-for-collaborative-ranking.pdf>

¹⁰⁷ Steffen Rendle et al. *BPR: Bayesian Personalized Ranking from Implicit Feedback*.

IF YOU HAVE IMPLICIT DATA

The algorithm we are talking about here is usually only used on implicit feedback, but here the problem is that you never have any negative feedback because you only have events that say "bought." Not having negative data is something that makes it hard for a machine learning algorithm to understand when it is doing wrong, so it also doesn't know when it's doing good. The absence of a "bought" event could show either that the user doesn't know it exists, or that the user knows about it but doesn't like it. In both cases we can assume that the *not* is something worse than *bought*. As illustrated in figure 13.6. Either a user has bought an item or not (the squared boxes) we then have different conditions the user-item relationship can be in. A user that hasn't bought it, it can be either because haven't seen it, seen it but didn't like it, or seen it, liked it but hasn't bought it yet.

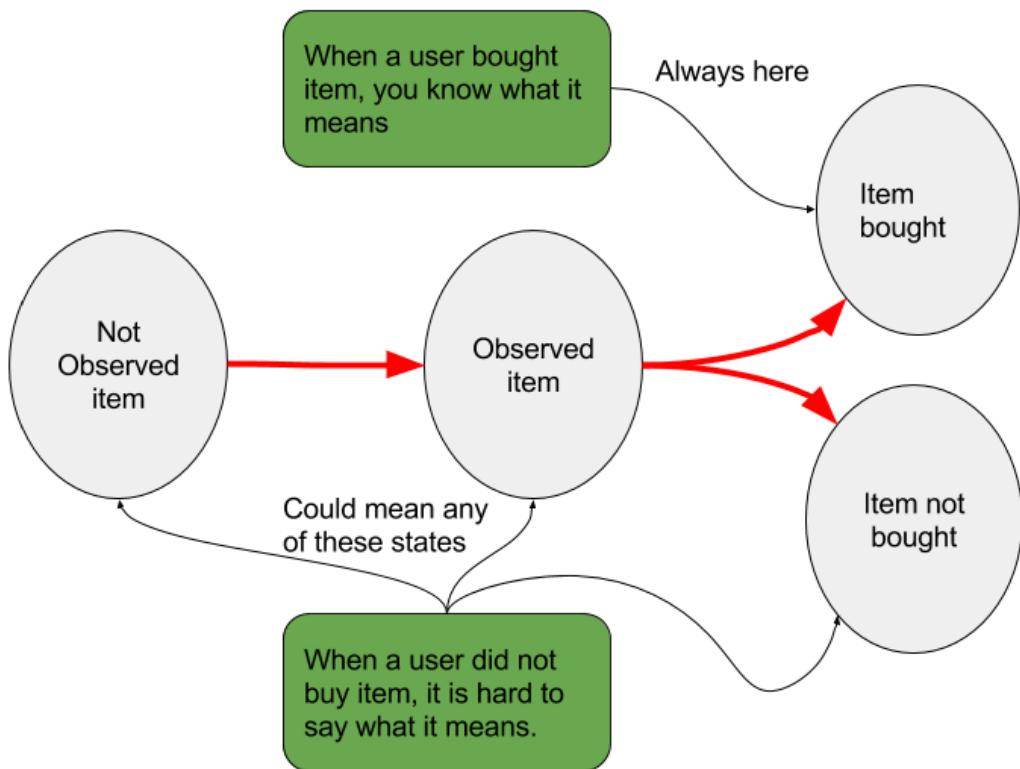


Figure 13.6 different states of a user-item relationship. We know that when a user bought an item it is bought, but if a user didn't buy an item, what does it mean then?

If we have two items, one that is bought and one that is not, then when talking about ranking, we can define that a bought item was always more attractive than one that hasn't been

bought. With that cleared out, we could now turn to item pairs that have two items that have neither been bought or that have both been bought – but we don't have anything we can say about them right now.

In figure 13.7 we can see the transformation of a user's buy log into an order matrix. Figure 13.8 shows how this will expand of our data since each user will have its own matrix

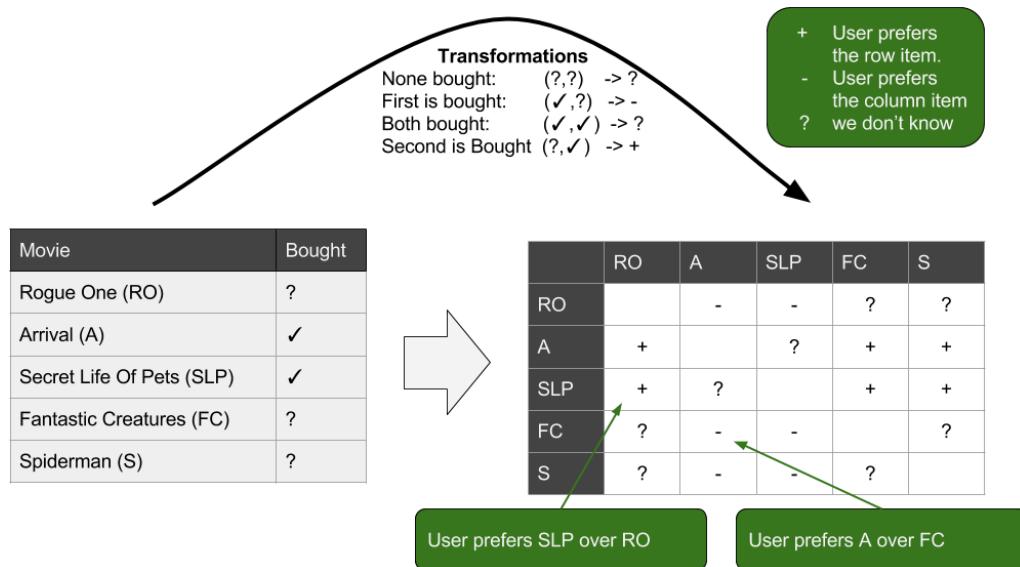


Figure 13.7 illustrating how to transform one user's transaction data into an order matrix. On the left “I” means that the user has bought the item. In the right side matrix, + means preference so for example, A is preferred over RO.

So how do we come up with this information? We're going to use an explicit dataset and a training dataset to do this.

WITH EXPLICIT DATASET

We can do the same ordering, saying that items that have not been rated are below the ones that have been rated. We could ask if the not-rated item should be valued as an average rated item or as an item that is rated below all rated items.

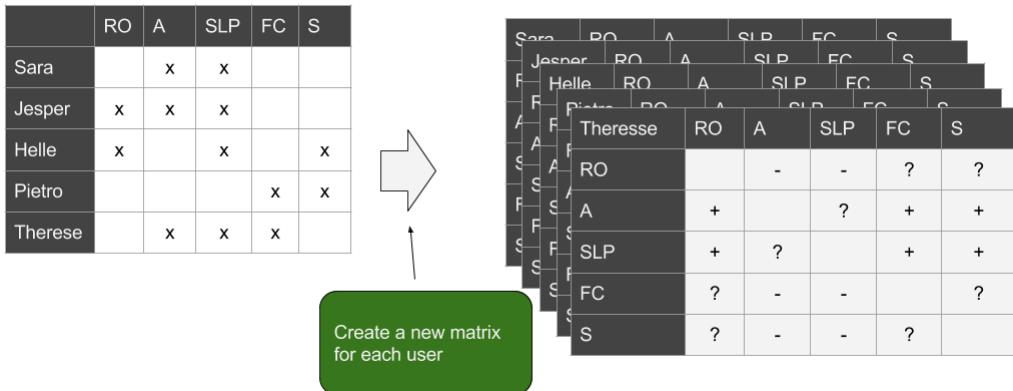


Figure 13.298 The algorithm creates an order matrices for each user. So 5 users in the rating matrix, will generate 5 matrices.

THE TRAINING DATASET

With the approach we have described above, we can now collect a data set to be used to train a ranking recommender. This data set will contain:

All tuples (u, i, j) where i has been bought/rated by the user, and j has not.

13.5.1 BPR

So, let's begin with the beast. We want to find a personalized ranking for all items and all users in our database. From the name, it is clear that we will use something called Bayesian statistics to solve this problem.

Bayesian statistics is all based on the following a simple equation:

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)}$$

Which states that the probability of event A happening given B happened is equal to the probability that A happens times B happens when A has occurred, divided by the probability that B happened. If we should do a real-life example we could say A is the event that it has rained and B is the event that the street outside is wet. Then Bayes says that the probability that it has rained given that the street is wet ($p(A|B)$) is equal to the probability that it has rained ($p(A)$) multiplied with the probability that the street is wet given it has rained ($p(B|A)$) divided with the probability that the street is wet ($p(B)$).

This simple equation has turned into a branch of statistics, which are really interesting, so I would just encourage you to look it up.

In the context of the Ranking problem we can formulate it by; having an unknown ordering preference from each user, which we denote by $>$ for user u . $>$ is a total ordering which given any two content items i and j from our catalog the user will prefer one or the other. We will also say that Θ is the list of parameters we need to find for a recommender system (or in fact any machine learning prediction model). For example, if we are talking about the FunkSVD, remember that the problem boiled down to make two matrices which you could use to calculate the predictions. Like the following:

$$\begin{bmatrix} 5 & 3 & 0 & 2 & 2 & 2 \\ 4 & 3 & 4 & 0 & 3 & 3 \\ 5 & 2 & 5 & 2 & 1 & 1 \\ 3 & 5 & 3 & 0 & 1 & 1 \\ 3 & 3 & 3 & 2 & 4 & 5 \\ 2 & 3 & 2 & 3 & 5 & 5 \end{bmatrix} = \begin{bmatrix} u_{1,1} & u_{1,2} \\ u_{2,1} & u_{2,2} \\ u_{3,1} & u_{3,2} \\ u_{4,1} & u_{4,2} \\ u_{5,1} & u_{5,2} \\ u_{6,1} & u_{6,2} \end{bmatrix} \begin{bmatrix} v_{1,1} & v_{1,1} & v_{1,1} & v_{1,1} & v_{1,5} & v_{1,6} \\ v_{2,1} & v_{2,2} & v_{2,3} & v_{2,4} & v_{2,5} & v_{2,6} \end{bmatrix}$$

When we talk about Θ in relationship to this then Θ is the set of all the $u_{i,j}$'s and $v_{i,j}$'s

Bayesian Personalized Ranking (BPR) we want to find a model, or more simply the Θ , as we just defined it, such that there is the highest probability that the model will produce a perfect ordering $>_u$ for all users.

The probability can be written like this

$$p(\Theta | >_u)$$

(you read it saying the probability of seeing Θ (theta) given the ordering $>_u$)

Due to Bayes we know that if we want to maximize this, then it is the same as maximizing the following because they are proportional¹⁰⁸.

$$p(>_u | \Theta)p(\Theta)$$

Notice that the ordering and the model have changed places. So now we will work on the probability of seeing the ordering $>_u$ given a specific model . And multiply that with the probability of seeing that model. We need to do some math magic in the following. Feel free to skip it if you are not interested in the nitty details. But before going into the magic, let's stay afloat for just a little bit longer and spell out exactly what is happening.

¹⁰⁸ https://en.wikipedia.org/wiki/Bayes'_theorem

We assume that in a perfect world there is a way to order all our content items perfectly for each of our users. Which is the total ordering \succ_u that I keep mentioning. If there is such an ordering, there is also some probability that there is a recommender algorithm that can produce it.

So $p(\Theta|\succ_u)$ is a question we are asking, assuming this ordering exists, what is the probability that we can find a model that will produce it. Then we mix Bayes into it, and we rephrase the question into. Saying that $p(\Theta|\succ_u)$ is the same as asking what is the probability that there is a Θ times the probability that if we have Θ , we then have the ordering. More clear now?

13.5.2 Math magic (advanced section)

We will look at the two parts of the expression above.

ASSUME THE PRIOR IS A NORMAL DISTRIBUTION (ADVANCED SECTION)

Let's start with the last part first ($p(\Theta)$), we will assume that the parameters of the model are independent and that each of them is normal distributed ($p(\Theta) \sim N(0, \Sigma_\Theta)$). Assuming that, we can write the last part as:

$$p(t) = \sqrt{\frac{\lambda}{2\pi}} e^{-\frac{1}{2}\lambda t^2}$$

if we say that $\Sigma_\Theta = \lambda_\Theta I$

LIKELIHOOD FUNCTION

Moving to $p(\succ_u | \Theta)$ we can do some rewriting. When we say \succ_u its only one user, but we want to maximize it for all users, so it means that we actually want:

$$p(\succ_{u1} | \Theta) * p(\succ_{u2} | \Theta) * \dots * p(\succ_{un} | \Theta)$$

Which can be written more compact as $\prod_{u \in U} p(\succ_u | \Theta)$. So we have a product of the probability that there is an ordering for each user, given such a model.

Which can be written more compact as $\prod_{u \in U} p(\succ_u | \Theta)$. So we have a product of the probability that there is an ordering for each user, given such a model.

The probability of a ordering for one user, must be the probability that for all pairs of items where one has been bought and the other one now, there is an ordering. We said above we only looking at these cases. With this common sense and some clever tricks we can reduce the product above to a product containing probability of an ordering for each data point (u, i, j) , meaning all our data D_s :

$$\prod_{u \in U} p(\succ_u | \Theta) = \prod_{(u,i,j) \in D_S} p(i \succ_u j | \Theta)$$

This we can take one step further, since Θ is a recommender system model, we know that $(i \succ_u j | \Theta)$ actually means that we are asking for the probability that there exist a recommender which will predict ratings such that $r_i - r_j > 0$. So we can rewrite the product again, into:

$$\prod_{(u,i,j) \in D_S} p(i \succ_u j | \Theta) = \prod_{(u,i,j) \in D_S} p(r_{ui} - r_{uj} > 0)$$

RELAXING THE ORDERING

Earlier I said that the problem with ranking was binary in the sense either it was item i that was preferred or not. And since we have the total ordering this describes a function which is called the Heaviside function, as shown in figure 13.9. Because the outcome of asking is $i \succ_u j$? can only ever be {yes, no} (yes or no), given a certain model. So $p(i \succ_u j | \Theta)$ is either 0 or 1.

We are only looking at the data where one item has been bought by the user and the other one hasn't. That means there is no sliding on the function. Its 0 until there is a straight vertical line to 1.

$$p(i \succ_u j | \Theta) = \begin{cases} 1 & \text{if } i \succ_u j \\ 0 & \text{otherwise} \end{cases}$$

Optimizing was analog to standing on a foggy hilltop and looking for water, then it doesn't work if the function one minute 1 and the next 0. you can't see which way to go down safely anywhere on a Heaviside function. So, to solve this, we cheat a bit and use another function which is almost the same, one called the sigmoid function. The sigmoid function also runs in the interval from 0 to 1 and moves almost as the Heaviside function. The sigmoid is defined as

$$\sigma(x) = \frac{1}{1 + e^{-(x)}}$$

Check out figure 13.9 (again) to see the sigmoid function in action.

As you can see from the figure, we can insert the sigmoid without losing too much integrity. So we get that

$$p(i \succ_u j | \Theta) = p(r_{ui} - r_{uj} > 0) = \sigma(r_{ui} - r_{uj}) = \frac{1}{1 + e^{-(r_{ui} - r_{uj})}}$$

Where:

r_{ui} : predicted rating from a recommender system.

We now have the building blocks to put everything together and come up with something we can stuff into some python code and do ranking.

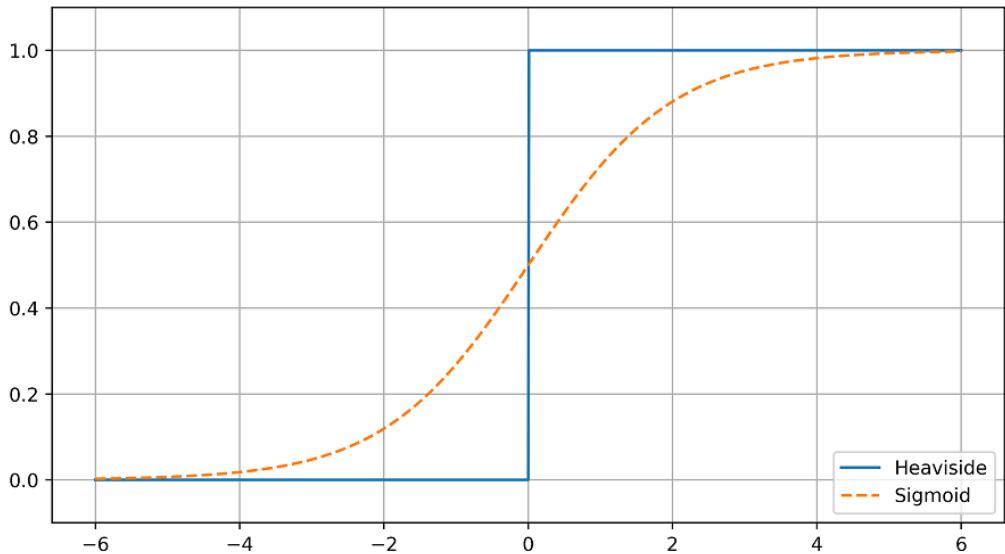


Figure 13.9: Heaviside and Sigmoid Functions plotted in the interval -6 to 6.

Once again, we want to find a set of parameters for a model such that we have the highest probability to produce a ranking for all users that is perfect. We can say we want to maximize the following (we write argmax when we want to say that we want to find the parameters that maximize the expression):

$$\underset{\theta}{\operatorname{argmax}} p(\theta | \succ_u) p(\theta) =$$

We will use a trick saying that if want to maximize that it's the same as maximizing the following, because the function \ln is continuous and always increasing:

$$\underset{\theta}{\operatorname{argmax}} \ln(p(\theta | \succ_u) p(\theta)) =$$

inserting what we deduced above we get:

$$\underset{\theta}{\operatorname{argmax}} \ln \left(\prod_{(u,i,j) \in D_s} \sigma(r_{ui} - r_{uj}) * \sqrt{\frac{\lambda}{2\pi}} e^{-\frac{1}{2}\lambda\theta^2} \right) =$$

Where:

D_s : is all the combinations we have in our data where a user u has bought/rated an item i but not the item j .

The function we added called \ln is short for the natural logarithm, and it has some nice properties¹⁰⁹ ($\ln(a*b) = \ln(a) + \ln(b)$) and $\ln(e^{-\frac{1}{2}\lambda\theta^2}) = -\frac{1}{2}\lambda\theta^2$) which we will use to simplify the expression above, at least a little bit.

$$\operatorname{argmax}_{\Theta} \sum_{(u,i,j) \in D_s} \ln(\sigma(r_{ui} - r_{uj})) - \lambda \|\theta\|^2$$

We will call the part inside of the argmax for BPR optimization criteria (BPR-OPT)

Okay. So this is the task. Did everybody arrive here, safe and sound? Let's recap. The expression we arrived at is what the problem boils down to. We want to find the recommender system, run with the set of parameters Θ will make the whole expression as large as possible, meaning that the it's the Θ with the highest probability that the system produces an ordering \succ_u which matches all user's preferences. It's a noble goal, don't you think?

I am afraid to say that this was only the problem, now we also need some more work before we get at the actual algorithm that solves it. Remember we used Stochastic Gradient Descent in chapter 10. We will use something similar here.

We want to find the gradient of the expression above; to understand which way we should move to get closer to the optimal place. I will claim (without proof) that the gradient of the BPR-OPT is proportional to the following (\propto mean 'proportional to'):

$$\frac{\delta BPR - OPT}{\delta \theta} \propto \sum \frac{e^{-(r_{ui} - r_{uj})}}{1 + e^{-(r_{ui} - r_{uj})}} * \frac{\delta}{\delta \theta} (r_{ui} - r_{uj}) - \lambda \theta$$

And this is the function we want to use to figure out which direction we should go, to optimize the ranking method.

With that we leave magic math mode again. And continue as nothing has happened and go through a way to optimize the expression found here.

13.5.3 The BPR algorithm

In the article where BPR is described they also suggest an algorithm called LearnBPR, which goes as follows:

- 1: procedure LearnBPR(D_s, Θ)
 - 2: initialize Θ
-

¹⁰⁹ https://en.wikipedia.org/wiki/Natural_logarithm#Properties

```

3: repeat
4: draw(u,i ,j)
5:  $\Theta \leftarrow \Theta - \alpha (\delta \text{BPR-OPT}) / \delta \Theta$ 
6: until convergence
7: return  $\Theta^*$ 

```

Soo... this is it. I bet it's a bit like you have watched a very complex film and then slept through the last ten minutes (where it was explained why it was the butler that did it). But we have actually arrived at an algorithm which will produce a ranking. The step in line five in the procedure (shown above) depends on which recommender we plug into it.

Most scientific articles use a matrix factorization algorithm to do it so we will do the same. The thing to ponder about is then if we use the same algorithm as we did in chapter 11 and the same heuristic to solve it why wouldn't it then produce the same results?

Perplexing I know, but now we have a new goal. Remember that in chapter 11 the goal was to reduce the difference between the ratings we have in our database and what the recommender could predict? Here we don't care what type of ratings are predicted, only the order of predictions, which allows the learning to be more "free" (for lack of a better word).

It is also worth mentioning the draw function, which can be done in several different ways and with different strategies. In the implementation, I did the simplest one, but there are other ways out there.

13.5.4 Bayesian Personalized Ranking with Matrix Factorization

If you don't remember what matrix factorization is, you can refresh your memory in Chapter 10. We will do a lot of the same things here.

Predicting a rating r_{uj} in matrix factorization comes down the multiplying a row in the user matrix W with a column in the item matrix H . Which is done by the following sum:

$$r_{uj} = \sum_{f=1}^K w_{uf} * h_{if}$$

Where K is the number of hidden factors.

To fit it into the expression above we need to consider how the gradient will look for this. We are taking the gradient in regards to Θ which is the union of all the parameters we are trying to find, meaning all the w 's and the h 's. But with some careful thinking you will see that there is only 3 cases that are interesting (as in non zero):

$$\frac{\delta}{\delta \Theta} (r_{ui} - r_{uj}) \begin{cases} (h_{ui} - h_{uj}) & \text{if } \Theta = w_u \\ w_u & \text{if } \Theta = h_i \\ -w_u & \text{if } \Theta = h_j \\ 0 & \text{else} \end{cases}$$

It requires some staring at the gradient expression above before realizing this. But I am afraid that I have to leave it as an exercise to the reader to do it.

13.6 Implementation of BPR

The BPR was first described in an article called “BPR: Bayesian Personalized Ranking from Implicit Feedback” which much of this chapter is based on. The authors along with some other people implemented a recommender system algorithm library called MyMediaLite¹¹⁰ in C#, the code you will see in the following is inspired by that. Figure 13.10 shows an overview of what we are implementing in the following

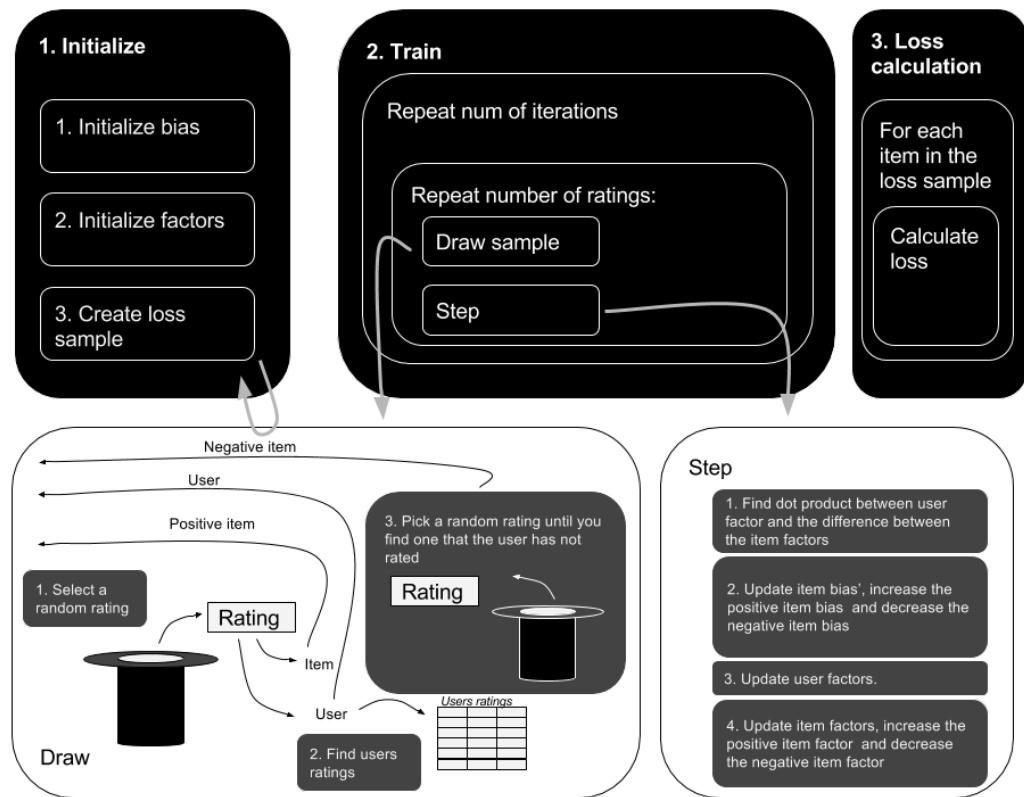


Figure 13.10 An illustration of what we implement in the following. We start by initializing everything. Then we train. For each iteration we run through the same number as ratings, for each step we draw a sample of a user

¹¹⁰ <https://github.com/zenogantner/MyMediaLite>

and a positive item and a negative item. Then we step, meaning that we move all the factors and bias' in the right direction.

To run the training you should of course have downloaded the code from github (<https://github.com/practical-recommender-systems/moviegeek>) and followed the install instructions there. Then you should move into the moviegeek folder and execute the following

Listing13.1: Running the bpr training algorithm

```
$> python -m builder.bpr_calculator
```

it will output something close to the following

Listing13.2: script output

```
2017-11-19 16:23:59,147 : DEBUG : iteration 6 loss 2327.0428779398057
2017-11-19 16:24:01,776 : INFO : saving factors in ./models/bpr/2017-11-19
16:22:04.441618//model/19/
```

To use this model, you need to take the folder name where the model (factors) has been saved and insert it into the recommender class, this could of course have been done automatically in a real system. But it is good to have a manual step in, so you are sure you don't have faulty models suddenly running in production. Insert path in recs/bpr_recommender.py in line 17 or as the default parameter to the init method.

Listing13.3: Init method of BRP recs – recs/bpr_recommender.py

```
def __init__(self, save_path='<insert path there>'): ①
    self.save_path = save_path
    self.load_model(save_path)
    self.avg = list(Rating.objects.all().aggregate(Avg('rating')).values())[0]
```

① insert path here. The log only prints out the relative path. But here you need the full one.

Before starting on the actual algorithm, we need to handle transform our rating data into something we can use. The BPR uses implicit feedback, which can either be clicks or purchases. if we consider the user-content lifetime which was described in chapter 4, then you could say that anything the user rated is something that they purchased. So, we could just say that all ratings are indications that the user bought something. The question then is whether we want to "loose" the information about whether a user has rated something high or not. If we want to take advantage of the user's explicit feedback we transform all ratings above a certain threshold indicates a buy and the rest we delete. It's a matter of gut feeling. In our case we will do the first solution, that way we will have more data to use.

In the description of the LearnBPR training algorithm line 4 you draw an example of data, which means to take a sample from your data. The draw consists of a user id, and two content ids. Where one content item is preferred by the user over the other one. This can be done by

simply saying that the preferred item is the one that the user has purchased, and the other one the user hasn't bought.

To draw a sample like that with our rating data you can simply do the following:

- Draw a random user rating, and you got the user id and the positive item
- Keep drawing random ratings, until you have a content item that is not rated by the user.

This of course leaves some assumptions about your ratings which you should remember. This dataset (MovieTweetings) only contains content if somebody has rated it, hence all the content is mentioned in the ratings data. Second you should consider that popular items will appear more frequently than others simply because they are rated more. This also has an influence on the drawing of data.

Listing13.4: draw method found in build\bpr_calculator.py

```
def draw(self, no):
    for _ in range(no):
        u = random.choice(self.user_ids)
        user_items = self.user_movies[u]
        pos = random.choice(user_items)
        neg = pos
        while neg in user_items:
            neg = random.choice(self.movie_ids)
        yield self.u_inx[u], self.i_inx[pos], self.i_inx[neg]
```

- ① the method should spit out a sample `no` times, so lets loop over the drawing a sample `no` times.
- ② select a random user
- ③ get the random users items.
- ④ below we want the while loop to continue until it has a found an item the user has not rated, setting `neg` to `pos` ensures us that it will enter the while loop
- ⑤ the while loop
- ⑥ randomly choose an item.
- ⑦ yield the sample.

First of all, we have the overall build method which is where we control it all. It looks like the following:

Listing13.5: build\bpr_calculator.py

```
def train(self, train_data, k=25, num_iterations=4):
    self.initialize_factors(train_data, k)
    for iteration in range(num_iterations):
        for usr, pos, neg in self.draw(self.ratings.shape[0]):
```

```
self.step(usr, pos, neg)
```

④

- ① initialize the factors
- ② loop the num_iterations times.
- ③ loop through all the samples create in the generate_samples method
- ④ Call the step method

If you were expecting the big light, then I guess this is a bit disappointing so let's move on quickly, But first, the method loops the number of times indicated in the num_iterations, and in each look it loops through all the samples of u,i,j which are users where item i was bought, and item j no. In our case, it means a randomly select. For each of those we call a step method, the step method does exactly the same as the one we implemented for the matrix factorization in chapter 11, so encourage you read through it but for any details please refer to the code in bpr_calculator.py and chapter 11.

Listing13.5: build\bpr_calculator.py

```
def step(self, u, i, j):
    lr = self.learn_rate
    ur = self.user_regularization
    br = self.bias_regularization
    pir = self.positive_item_regularization
    nir = self.negative_item_regularization
    ib = self.item_bias[i]
    jb = self.item_bias[j]
    u_dot_i = np.dot(self.user_factors[u, :], self.item_factors[i, :] - self.item_factors[j, :]) ③
    x = ib - jb + u_dot_i
    z = 1.0/(1.0 + exp(x))
    ib_update = z - br * ib
    self.item_bias[i] += lr * ib_update
    jb_update = -z - br * jb
    self.item_bias[j] += lr * jb_update
    update_u = ((self.item_factors[i, :] - self.item_factors[j, :]) * z
                - ur * self.user_factors[u, :])
    self.user_factors[u, :] += lr * update_u
    update_i = (self.user_factors[u, :] * z
                - pir * self.item_factors[i, :])
    self.item_factors[i, :] += lr * update_i
    update_j = (-self.user_factors[u, :] * z
                - nir * self.item_factors[j, :])
    self.item_factors[j, :] += lr * update_j
```

- ① short nicknames the the learning rate and regularization constants.

- 2 same with the item bias
- 3 taking the dot product between user factor and the difference between the two item vectors.
- 4 update the item bias'
- 5 update the users factor vector
- 6 update the item factors

What is more interesting in this chapter is how the sample is done, and how the prediction and loss function looks. First the drawing of samples. The sampling method should pick a user with one item bought and one not bought. The draw method below uses yield instead of return, this means that when it arrives to yield then it delivers the result, but it will stay in the for-loop, such that it will iterate through all the index. This could also have been done simply by filling all the samples into a list, and then return the list. Yield just seems a nicer way of doing it.

Listing13.6: Build\bpr_calculator.py

```
def draw(self, no=-1):
    if no == -1:
        no = self.ratings.nnz
    r_size = self.ratings.shape[0] - 1
    size = min(no, r_size)
    index_randomized = random.sample(range(0, r_size), size)
    for i in index_randomized:
        r = self.ratings[i]
        u = r[0]
        pos = r[1]

        user_items = self.ratings[self.ratings[:, 0] == u]
        neg = pos
        while neg in user_items:
            i2 = random.randint(0, r_size)
            r2 = self.ratings[i2]
            neg = r2[1]

    yield self.u_inx[u], self.i_inx[pos], self.i_inx[neg]
```

- 1 We want to shuffle our data, so create an array of all the numbers in the index (0 till the end) and then we shuffle it.
- 2 run through the shuffled index.
- 3 a rating is selected, now find all the users ratings (here is a place where you could probably optimie the code, so you don't have to filter all the ratings every time)
- 4 Silly trick to get through the loop constraint the first time.
- 5 loop until neg is a content item which the user hasn't rated.

The loss function is the one that indicates if you are going in the right direction. It finds the ranks of the two items and calculate the $\sum_{(u,i) \in D_S} \ln(\sigma(r_{ui} - r_{uj})) - \lambda \|\Theta\|$.

Listing13.7: Build\bpr_calculator.py

```
def loss(self):
    br = self.bias_regularization
    ur = self.user_regularization
```

```

pir = self.positive_item_regularization      ①
nir = self.negative_item_regularization     ①

ranking_loss = 0
for u, i, j in self.loss_samples:
    x = self.predict(u, i) - self.predict(u, j)
    ranking_loss += 1.0 / (1.0 + exp(x))      ②
    ②
    ②

c = 0
for u, i, j in self.loss_samples:           ③
    c += ur * np.dot(self.user_factors[u], self.user_factors[u])
    c += pir * np.dot(self.item_factors[i], self.item_factors[i])
    c += nir * np.dot(self.item_factors[j], self.item_factors[j])
    c += br * self.item_bias[i] ** 2
    c += br * self.item_bias[j] ** 2

return ranking_loss + 0.5 * c               ④

```

- ① create short nicknames for the constants
- ② calculate the ranking loss
- ③ regularization expresions.
- ④ ranking loss plus half the regularization.

The loss function uses the prediction method, the difference with the predictions is that it is not a rating that is predicted it is a value compared to the prediction of another item, shows how those two should be ranked against each other.

Listing13.8: Build\bpr_calculator.py

```

def predict(self, user, item):
    i_fac = self.item_factors[item]
    u_fac = self.user_factors[user]
    pq = i_fac.dot(u_fac)                      ①

    return pq + self.item_bias[item]            ②

```

- ① do the dot product between the item factors and the user factors.
- ② add the item bias and return.

Running this algorithm takes a long time. There are a lot of places that you could be more smart and probably you could optimize it to run several 100 times faster with a few tricks. But as it stands here it takes my MacBook 2017 model around two hours pr iteration. So 20 iterations is 40 hours. So here you can really go for a run or something. When finished we can use it to make recommendations, let's look at how you do that in the following

13.6.1 Doing the recommendations

Listing13.9: recs\bpr_recommender.py

```

def recommend_items_by_ratings(self, user_id, active_user_items, num=6):

    rated_movies = {movie['movie_id']: movie['rating']
                    for movie in active_user_items}          ①

```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/practical-recommender-systems>

Licensed to Asif Qamar <asif@asifqamar.com>

```

recs = {}
if str(user_id) in self.user_factors.columns: ②

    user = self.user_factors[str(user_id)]
    scores = self.item_factors.T.dot(user) ③

    sorted_scores = scores.sort_values(ascending=False) ④
    result = sorted_scores[:num + len(rated_movies)] ⑤

    recs = {r[0]: {'prediction': r[1] + self.item_bias[r[0]]}
            for r in zip(result.index, result)} ⑥
            if r[0] not in rated_movies}

    s_i = sorted(recs.items(),
                 key=lambda item: -float(item[1]['prediction'])) ⑦

return s_i[:num]

```

- ① Make a dictionary of the active users movies, it comes in handy when verifying that we are not recommending anything the user has already seen.
- ② Be sure the model has seen the user, otherwise we can't return any recommendations
- ③ Calculate the dot product between active user factor and all the item factor vectors. Such that we can calculate which are more alike.
- ④ order values descending
- ⑤ cut the list down to the number that should be returned plus the number of ratings the user has
- ⑥ Run through the resulting items, adding the item bias
- ⑦ order again, and return expected numbers.

To do this we first have to load the model, which is saved in the last step of the training. We load it using the following.

Listing 13.10: recs\bpr_recommender.py

```

def load_model(self, save_path):

    with open(save_path + 'item_bias.data', 'rb') as ub_file:
        self.item_bias = pickle.load(ub_file)
    with open(save_path + 'user_factors.json', 'r') as infile:
        self.user_factors = pd.DataFrame(json.load(infile)).T
    with open(save_path + 'item_factors.json', 'r') as infile:
        self.item_factors = pd.DataFrame(json.load(infile)).T

```

13.7 Evaluation

How do we test that? Well there is the offline evaluation which you learned about in chapter 9. Where you do cross validation. As shown in figure 13.12

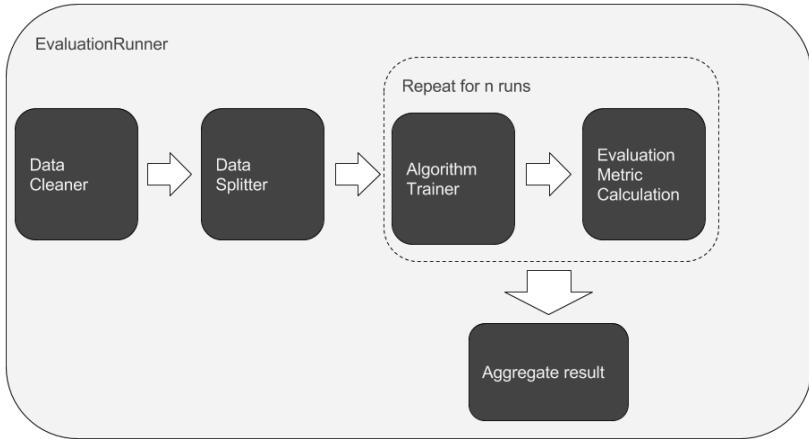


Figure 13.11 The evaluation runner is used to evaluate an algorithm. It is a pipeline where the data is first cleaned, then split into the k folds for cross validation. Then for each fold it repeats training of the algorithm, evaluating the algorithm. And then finally when it is all finished, you aggregate the result.

As a small reminder, I added the evaluation method that runs that creates the data which is shown in graph seen below. Listing13.11: evaluator/evaluation_runner.py

```

def evaluate_bpr_recommender():
    timestr = time.strftime("%Y%m%d-%H%M%S")
    file_name = '{}-bpr-k.csv'.format(timestr)

    with open(file_name, 'a', 1) as logfile:
        logfile.write("rak,pak,mae,k,user_coverage,movie_coverage\n")

        for k in np.arange(10, 100, 10):
            recommender = BPRRecs()
            er = EvaluationRunner(0,
                None,
                recommender,
                k,
                params={'k': 10,
                         'num_iterations': 20})
            result = er.calculate(1, 5)

            user_coverage, movie_coverage =
            RecommenderCoverage(recommender).calculate_coverage()
            pak = result['pak']
            mae = result['mae']
            rak = result['rak']
  
```

Measuring the precision gave the result shown in figure 13.12, which is nothing special. I have confidence that it can become better, they say it's good, but on small K it is not great. I also calculated the coverage and it is also better. Item coverage is 6.4 %, and User coverage is 99.9%

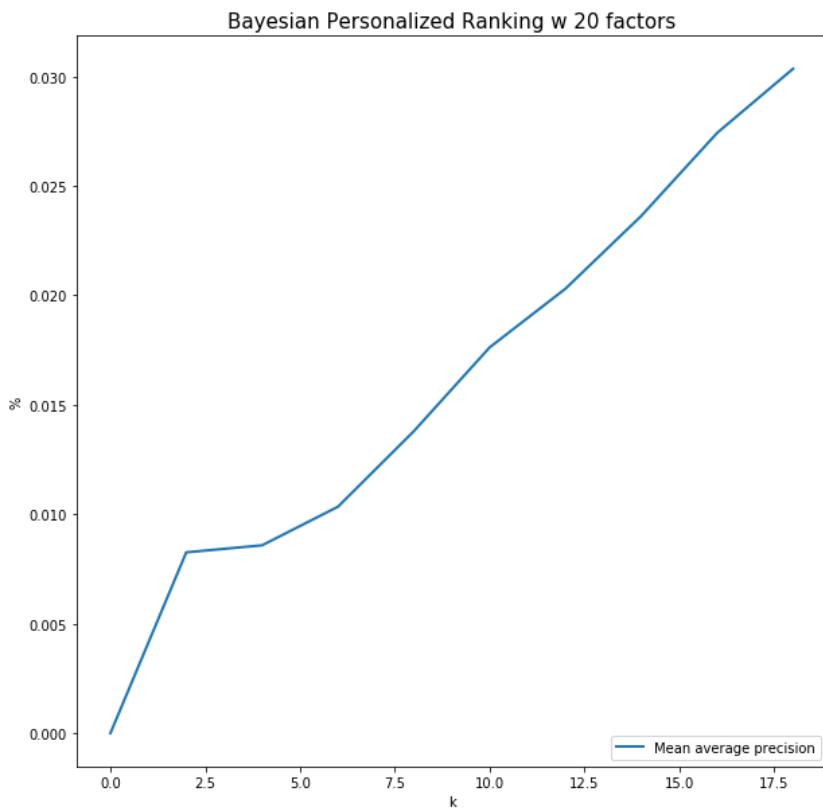


Figure 13.12 Mean average precision for BPR on a short top N. It is not that impressive, but I am sure it can be tweaked better than that.

If you look at larger numbers, then it suddenly looks much better, as shown in fig 13.13

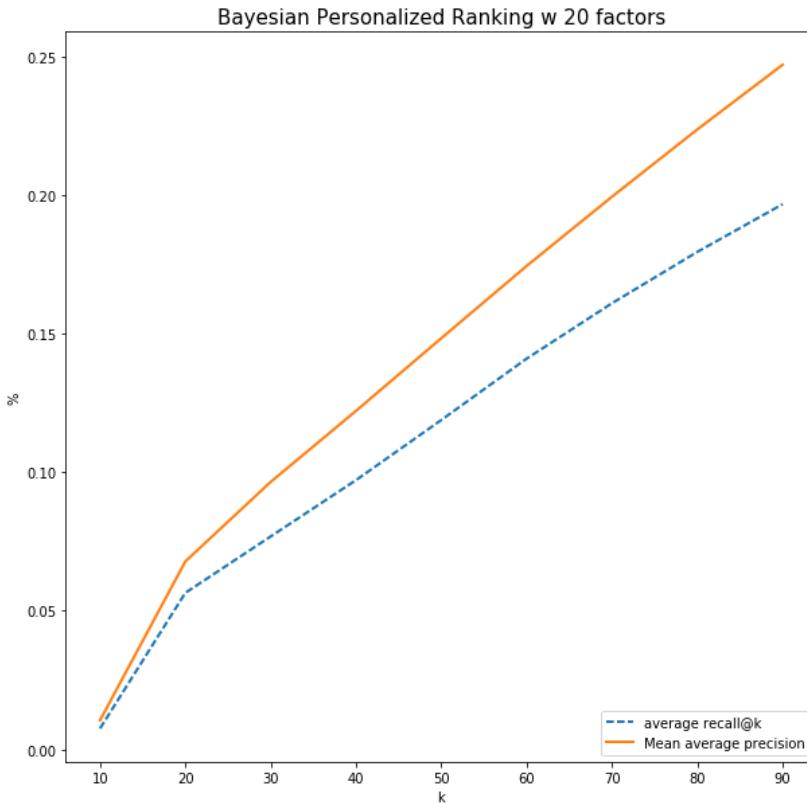


Figure 13.13 precision and recall for the BPR algorithm.

The thing about this graph and these numbers is that they are local to this dataset, and you can't really use them for much other than as a benchmark which can be improved upon. Again is that this recommender actually only recommends 6 percent of the items. Which is not a lot, so you should probably mix it with the content based recommender or use something else like the multi armed bandit scheme to introduce more item into the system.

13.8 Levers to fiddle with for the BPR

BPR is a complex algorithm and there are many decisions that has been done before running it, we glossed over a lot of them, for example how many factors should be included, what learning rate enables the system to optimize the learning problem best possible. That doesn't mean they are not important here, only that it is up to you to use what you learned in former chapters to evaluate the meta parameters.

But let's take a quick walk through of them. We have the item bias and the user factors, we need to decide how many factors we want to use. The number of factors should be decided by how complex your data domain is. For example, movies are overall into a small set of types of movies, so maybe they don't need too many factors. While if you are making a wine recommender like the Vivino.com one, then you probably need a lot more. This is something you should test based on your dataset.

The learning rate is hard to give good advice about, I found that a too high learning rate gave me very large user factor values, meaning that the bias' didn't have a lot to say. Which a too low learning rate would allow the bias' to take over the decisions. There are also the regularizations which then try to keep things a bay, but if they need to be too large then maybe it's a sign that the learning rate is too high.

I love the idea of comparing items and then pushing them in different directions (I am talking about the item bias' here). This can be adjusted with positive and negative item regularization. These will also have an effect on how much a negative item is pushed away, and that might hurt new items if there isn't a lot of users who has consumed it yet.

The data that we have used had ratings, we could have taken the low rated items away, so they didn't figure as positive items in our training set, but on the other hand, even if a use didn't like the movie, it meant that it was something that they had consumed. So, it is good to use those also.

13.9 Summary

Did you learn everything you needed? If not then you can start over on this chapter again, but it is pretty hard. and unless you are going to implement the BPR yourselves, maybe you don't need to remember all details. But if you are planning to implement a recommender using BPR then it's a good idea to know how it works. Learning to rank is also something you will have to consider when you work with search machines, so it's not a bad thing to know a little bit about.

- LtR algorithms try to solve a different problem to the one of the classic recommender problem to predict ratings.
- Foursquares have used a ranking algorithm to combine the ratings and the locations of the venues. Which is a great example of how to combine relevant data into one recommendation.
- The *Learning to Rank* has the challenge that it is not easy to come up with a continuous function which can be optimized.
- In this chapter, we touched upon the Bayes Theorem, even if it is not the subject of

this book, you should look into it¹¹¹, as it is used in many scenarios.

- Using the Bayes Personalized Ranking (BPR) can be used on top of the matrix factorization method we looked at in chapter 10, but also other types of algorithms.

This is it. One more chapter to go, and you can go to GoodReads.com and mark this book as read. Remember to review it where you bought it, good or bad, recommender systems need feedback, so please Support that.

¹¹¹ Here is a book about probabilistic programming where they talk more about using Bayes: <https://www.manning.com/books/practical-probabilistic-programming>

14

Future of Recommender Systems

The last chapter of this book will covers:

- You will have a short book summary
- I will provide you with a list of topics to learn next, if you want to continue your voyage into the exciting world of recommender systems
- Nobody knows what the future of recommender systems is, but I will give you my best bet.
- And finally the final thoughts.

It has taken me two years to arrive at writing this sentence; I hope you travel time has been a bit faster. I wish I could say that now you know everything about recommender systems and that you can go into the world and understand all recommender algorithms and never to be surprised by anything on this topic again. You have come a long way, but from here to being an expert is still a very long way.

In this book, you learned the basics, enough to get you started and equip you for digging further into the subject. But don't just read. Play around with new algorithms or get more intimately familiar with the ones described in the book. There are many improvements and tricks to make each of them work better. Hopefully, the MovieGEEK site will have provided you the basics for how to load different kinds of data, and try out similar things. Just remember that MovieGEEK is implemented to make it easy to understand, and has lots of places where it can be optimized.

But before you go out into the world of recommender systems alone, I want to talk about a few things that didn't fit in this book. My editor would probably call it the cliffhangers for the next book, but my wife says no way – so you will probably be on your own from here.

First, let's have a quick run through of what topics we looked at in the book.

14.1 This Book in a Few Sentences

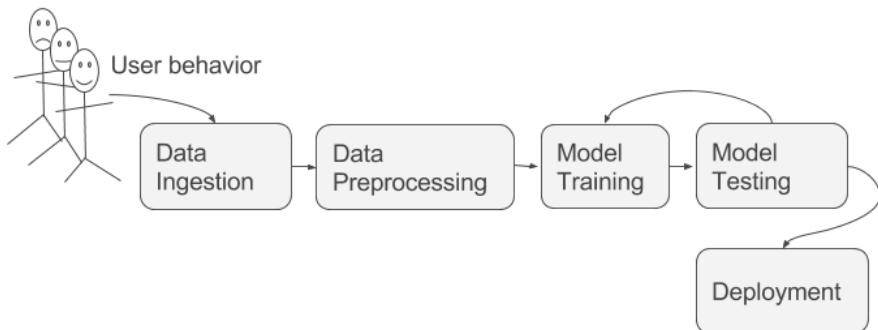


Figure 14.1 A recommender system pipeline

In this book, we have talked about recommender systems, which can be described by the pipeline shown in Figure 14.1. We started out talking a lot about collecting data or *data ingestion*. In Pedro Domingo's famous paper called "A Few Useful Things to Know about Machine Learning"¹¹² he points out that more data beats a more complex algorithm. That is also true for recommender systems, only data collected is not always a source of truth, because explicit ratings can be a reflection of mood or a result of social influences and not always be trusted to indicate what the user wants. Implicit ratings are, as the name implies, implicit and you or the machine have to deduce what the collection of events occurred between each user and item actually indicate. And finally, people's tastes change over time, so old data might be misleading. In other words, it is not always straightforward to understand what the behavioral data means. Turning events into ratings and clustering users are part preparing the data for creating a recommender model and are usually referred to *data preprocessing*.

While data is important for understand what the user wants, it is equally important to have a way to see how it is going, that is why we looked at some analytics that will be useful to keep an eye on how things are going.

¹¹² <https://homes.cs.washington.edu/~pedrod/papers/cacm12.pdf>

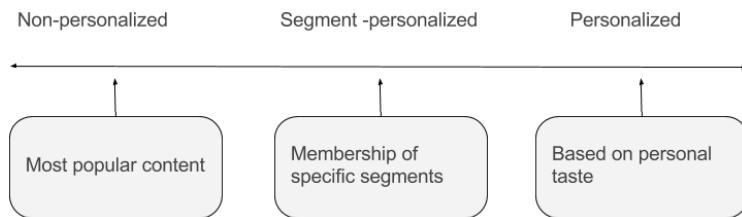


Figure 14.2 Grades of personalization

Calculating recommendations often uses a model and is covered by the *Model Training*. You can divide the different types of recommendations into various levels of personalisation, from completely non-personalised to very personalized, as shown in Figure 14.2. We started with the non-personalised as they don't require any user knowledge or frankly much other, this enables you to have a soft launch into adding recommendations to your site.

All users start out as cold users so we began there (chapter 6), and moved into looking at how you can segment things (chapter 7), so do semi-personalized or demographic recommendations. First there were the non-personalized recommendations which were about finding the most sold items, the most liked or trending items. Then we spend a chapter on one of the more severe problems in the recommender systems which is to figure out what to do with new users and items. An attempt to solve the recommender problem for new users is to looking at shopping basket analysis.

Then we looked at distance and similarity measurements between items and between users, which is something that haunts basically every recommender system algorithm. The first personalized recommendation method was collaborative filtering (in chapter 8), which provides recommendations based on similar behaviour either between users or items. This can be done using either ratings explicitly inserted into the system by the user or the implicit logging data. There are two types of collaborative filtering; we started with the one known as the neighbourhood based, the algorithm uses similarity measures to find neighboring items or users to the active one. With one algorithm under the belt, we had a look at the evaluation of algorithms (in chapter 9). We looked at many different metrics for evaluating a recommender and finally implemented the Mean Average Precision in the system.

If you don't have a lot of user data (or you have some well-described content), then it's worth to consider content-based recommenders, there are several ways to do that, but the core is that you look at the content, and calculate similarities on that basis. We looked at creating vectors using first TD-IDF and then topic vectors using Latent Dirichlet Allocation (LDA) topic modelling (in chapter 10).

After the LDA we return to collaborative filtering again, only now we looked at the model based. There are many different ways to do model based, but king of them is the matrix factorization way (chapter 11). This is also where we started looking seriously at training

machine learning algorithms. We talked about the traditional SVD and then moved on to the FunkSVD, which was used in the Netflix Prize competition.

With a good toolbox of different recommender algorithms, it was time to start looking at how to combine them, which you can do in many different ways. We explored some in the chapter on hybrids, where we implemented one called Feature Weighted Linear Stacking, which was the method that actually won the Netflix prize. Next, we looked at a new type of algorithm called Learning To Rank. Which doesn't care much about correctly predicting ratings, rather it focuses on producing lists of items which are ranked correctly. The algorithms can be divided into three different types as shown in figure 14.3

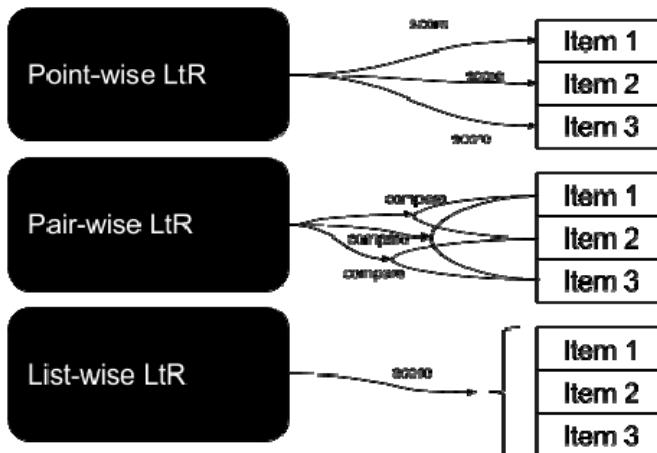


Figure 14.3 different types of learning to rank

There are advantages and disadvantages with all of them, I picked one that is often referenced when talking about learning to rank, the Bayesian Personalized Ranking algorithm.

At this point you might ask which of the algorithms should you start out implementing? It depends a lot on what kind of data you got. I would suggest going with the Matrix factorization. When that is implemented, you can build the learning to rank on top of it. Or you can add another implementation making it into an Ensemble.

And then finally we have arrived here at the future of your recommender journey and the future of recommenders in general. Let's start with the first one.

14.2 Topics to learn next

Here is what I recommend as the next steps in your pursuit to master recommender systems.

14.2.1 Further readings

First there is so much more research to dive into if you want more details and more ways to do recommendations. I can heartily recommend the book *Recommender Systems Handbook* by F. Ricci. It's a brick of a book, but it gets around a lot more than what I managed to fit into this book.

I had really hoped to fit more about online testing into this book. I did get to talk a bit about A/B testing in chapter 9, but the next topic you really need to learn about is exploit/explore methods and multi-armed bandits. For that purpose, I recommend that you take a look at the book *Statistical Methods for Recommender Systems*¹¹³ written by Deepak Agarwal and Bee-Chung Chen from LinkedIn.

Beyond books, keep an eye on [GroupLens](#)¹¹⁴, and generally for research look for the ACM Rec Sys conference. A lot of interesting papers come out of that, and YouTube videos, they have their own channel¹¹⁵. Reading this book you might get the picture that Collaborative filtering like it is done in chapter 8 is largely a historical approach, but it is still an approach where a lot of research is done, and the way lot of companies produce their recommendations. For example the paper of the year at RecSys2016 was an optimization to item-item collaborative filtering, the paper was called "Local Item-Item Models For Top-N Recommendation"¹¹⁶. You can catch Evangelia Christakopoulou, the author, on youtube talking about it¹¹⁷.

Recommender Algorithms are a subfield of Machine Learning. Few people will get away working with only the algorithms deemed recommender algorithms, I would, therefore, recommend that to continue your journey into becoming an expert in delivering relevant recommendations, by studying machine learning in general, beyond what we talked about here. Keeping within Manning a good start could be to look at the *Real-World Machine Learning* book and *Algorithms of the intelligent web*, both are quite accessible.

Search engine is a topic which is closely related to recommendations. My view of search engines is that they are seeded recommendations, so Search are a special case of recommender systems, while Search engine people like Dough Turnbull would say that recommenders are actually search machines. This could quickly become a discussion like the one about which came first the chicken or the egg. Either way it would be good to know something about that. Most places where recommenders are implemented there is also a search index to manage. Take a look at Doughs book *Relevant Search* for more thoughts on that.

¹¹³ <https://www.linkedin.com/pulse/practical-guide-recommender-systems-deepak-agarwal>

¹¹⁴ <https://grouplens.org/>

¹¹⁵

¹¹⁶ <http://glaros.dtc.umn.edu/gkhome/fetch/papers/gislimRecSys2016.pdf>

¹¹⁷ <https://www.youtube.com/channel/UC2nEn-yNA1BtdDNWziphPGA>

14.2.2 Algorithms

Turning now to algorithms, then *Learning to Rank* algorithms are quite popular, also the list-wise ones, which are a bit more complex than what we looked at in chapter 13.

Nowadays every serious machine learner will look to deep learning for improvements in all areas, and there is also intensive research going on in the subject of recommenders. Things are going so fast that it will be foolish to add any pointers here as I am sure it will be terribly outdated before this book is even printed. But take a look at the deep learning workshop at the rec sys conference¹¹⁸

One of the issues with the algorithms described in this book and one of the largely unresolved problems of recommender systems is that most recommenders are optimized to show items that are as similar as possible to items already seen by the user. But what we really want a recommender to do is find items, which are undiscovered, serendipitous and novel. It depends on which domain how much you need this, but it is worth thinking about, add a little bit of chaos into the recommendations also. A similar problem is also that people might not want to see the same top-N list of recommendations, over and over again.

14.2.3 Context

Recommender systems are moving from being just something that you see on your monitor to devices that can be in motion. Users could have one session in Europe and the next one in the US. Being something that is moved around it can also be influenced by the elements, temperature, rain, and other things. We have discussed this idea throughout the book, but without providing any concrete solutions. Depending on the domain you are in, it is worth also considering this. In Chapter 11 we talked about the FunkSVD that can be extended to also handle context. Or the reranking method described in chapter 13, could also be used.

14.2.4 Human Computer Interactions

As a data scientist or ML nerd, you probably would like to forget about anything regarding the front-end, but to do a good recommender you also need to serve it in the best way. User interface is important¹¹⁹

14.2.5 Choosing a good architecture

Let's face it. The MovieGeek site does not perform very well. With one concurrent user, it is okay, but I fear that it wouldn't hold much more!. It is, therefore, a good idea to look for a bit

¹¹⁸ This is the one done in 2016: <http://dlrs-workshop.org/dlrs-2016/program/> and they just announced one again for 2017. If you come to the RecSys conference, please say hi!

¹¹⁹ <https://www.quora.com/How-important-is-the-user-interface-for-a-recommender-system>. Another question is whether the way items are recommended needs to be the same for all people. People have different tempers, so maybe your recommender needs to be more discrete with some people and more in-your-face with others.

stronger platform to run a recommender and website on. Django is very performant, but it needs a real database. I have used SQLite because that is what requires the least setup, but I would recommend upgrading that to a PostgreSQL or similar before putting your site live. But in saying this, I do not intend that you should start out moving everything to a new architecture and buy a lot of new hardware. Try to figure out if a recommender will provide you with what you want at first. If you can't test it on all your traffic at first, it doesn't matter, create something smaller and show it to only 1% of your customers, and see how they react. Only remember that the more data you give a recommender, the better it works (this is as a general rule) But let's imagine that you have this step sorted already and you want to roll it out full scale, then here are some things that I would consider.

THE CLOUD DOESN'T SOLVE ALL YOUR PROBLEMS

We are all bombarded with ads about how one cloud service after another, beats all the others in being the best fastest and cheapest. But before moving things up there, please consider the following:

- Data needs to be in the cloud to be usable. That means that if you have an on-premise solution (meaning that you have your servers locally), then you will have to somehow move all your data into the cloud for it to do the calculations necessary to create the recommendations. Bandwidth, storage, etc. can quickly turn a cheap solution into something entirely costly.
- The off-the-shelf recommenders like Microsoft Cognitive Services Recommender API¹²⁰ are probably an easy start, just remember that there might be restrictions on how many items you can have in your catalog or how often it can be called. Be careful to research if those restrictions make it feasible to make your application or site rely on it.
- Privacy is something that should be taken very seriously, if you have any sensible data then remember that in the Cloud means "somewhere in the world," and it is the country where the server is placed which dictates the laws that are applied to the data stored on it.
- And finally, data is your most valuable asset; don't show it to others before considering what values you are providing to your competition.

On the plus side, a cloud service takes away a lot of the pressure of keeping a server running and scaling up in peak periods, so I am not saying you should disregard it, just consider the circumstances around it.

¹²⁰ <https://docs.microsoft.com/en-us/azure/cognitive-services/cognitive-services-recommendations-quick-start>

WHAT PROCESSING PLATFORM TO CHOOSE

And here finally it comes... ta da... SPARK. There is so much hype on Spark that this wouldn't be a good data book if I didn't mention it at least once.

Spark is a distributed computing platform which can handle a lot of the machine learning in a distributed very performant way. If you are interested in doing the calculations across many computers, then Spark is an excellent choice. If you look at Spark.Mlib¹²¹ it is possible to make the matrix factorization we talked about in chapter 11, in fact, it is described in an hands-on exercise from the Spark Summit 2014¹²². If you want to do a content-based recommender like the one from chapter 11, there is an implementation to do an LDA model, but I haven't found a tutorial to provide you. But keep your eyes on my blog kimfalk.org, it might appear there soon.

14.3 What is the future of recommender systems?

Predicting the future has proven to be quite difficult, and only the ones that predict it correctly will be remembered, so let's hope the following will be remembered, but remember the chances are that the future is entirely different.

One thing I am quite certain about is that recommenders will be something that will pop up everywhere. It will be the new JavaScript for the web and apps. They will run on everything, and provide their humble service in virtually any decision-making process.

USER PROFILES



Figure 14.4 A user profile could contain which movies and books the user likes, but also information

¹²¹ <https://spark.apache.org/mllib/>

¹²² <https://databricks-training.s3.amazonaws.com/movie-recommendation-with-mllib.html>

like that he is a parent, house owner and loves bikes.

In Denmark, we have had a lot of discussions about the fact that everything is becoming electronic and accessible only online. For example, the Danish state has stopped sending out paper letters and will only communicate using an electronic mailbox. State decisions and the light speed innovation doesn't, however, make all people in Denmark inclined to jump on the bandwagon and roll with it. This means that there will be an increasing gap between the people who are on the bandwagon and people who are not. Why is this important in this context? Well, you see people on the bandwagon are the people we recommender system engineers like because they are the ones that give us lots of evidence to work with. Tracking and knowledge about people that uses the net are only going to increase, which will mean that we will be able to get a pretty good picture of what they like. Facebook and LinkedIn are teaching people to have a public human readable profile. I think in not some distant future either Facebook or some other company will provide a public machine-readable profile for machine learning algorithms. Having such a profile will enable the systems to access and use it to understand who they are dealing with, this will work well for the entertainment industry, by having lists of movies or books which the person likes. But it might also contain segment details like house owner, car owner or parent as shown in figure 14.4, which will help other types of recommenders. A machine-readable profile like that will also enable humans to be much more conscious about whether you want to share your information or not. Currently, sites like Facebook have convinced people that it is a good idea to use your profile for logging in to different sites, without doing a lot of effort to inform you that you are actually allowing the company to access your profile, so in a certain sense this profile is already there. Having a format for a machine-readable profile can also enable better privacy because you can decide yourself how much you want to reveal about yourself. Letting a system know things like you are a car owner, you just read the latest Peter F. Hamilton Novel, and were crazy about new trends in organic raw food dishes.

For people not allowing access to their details, there are the companies in the business of generating profiles about users, and they are on the rise, which means that no matter what you do, then a business will probably know something about you from day one.

On the other hand the people not on the net, they will probably be even harder to serve recommendations to. But as the new generations will grow up, not knowing how it was before everybody had a smartphone, this segment will probably grow smaller by the year. But in the near future, it will still be a lot of problems with cold customers.

CONTEXT

Devices and choice is also becoming ever more dynamic. I believe that recommenders will be used in many different contexts, not only because almost all devices are now portable, with GPS and other tools to understand the current context, but also because soon all devices will contain so much content and choice that people will need a recommender to figure out what to do. Now I am not just talking about buying stuff. Recommenders will also be used to take

decisions in many other scenarios. Next Best Action recommenders are becoming bigger and bigger in marketing and banking. This is not necessarily for the end user, but to help bankers suggest to customer's what options are better for them, or lawyers to handle a case, or even for helping you finding the love of your life.

Current research and most public available knowledge about recommenders are about algorithms which do all the recommendation calculations offline. This fits with doing research and allows you to validate the algorithm on a test set, which can then be compared by other researchers who will try to make a better result. As mentioned several places in the book, this doesn't actually guarantee you will get produce a good recommender using this approach. I, therefore, think that the future of recommenders lies in dynamic recommender algorithms, of course with a core of offline calculations similar to the ones we have learned, but the idea of reinforced learning will have a much more important role.

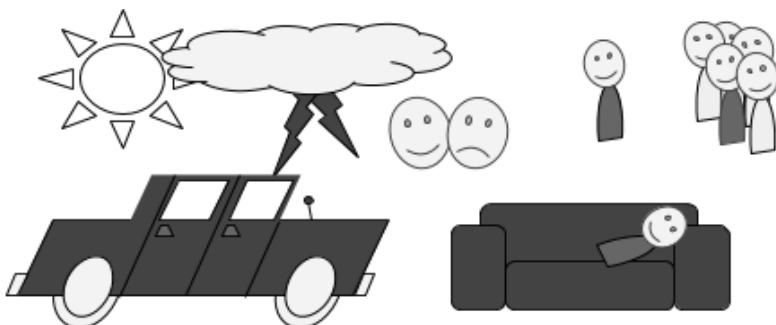


Figure 14.5 Context can be many things; the weather can have an impact, is the user happy or sad, alone or in company, or maybe driving or just home lying on the sofa. Each would mean different things in different domains.

The context could also be that your phone is connected to your smart watch which looks at your body functions and for example predicts that you need a drink because you are low on fluids, so it recommends a juice bar around the corner or it will play music that fits with your heart rate. Maybe a recommender will become a general recommender, humans like you, are in this situation also liked having a gin and tonic. For more indebt discussion about context-aware recommender systems please refer to the recommender systems handbook (or search for it on the net, as the chapter on context-aware recommenders is also available there)

ALGORITHMS

When I first started working on recommender systems, we would think of them as the natural evolution of search engines. Everything would be so data overloaded that everything would be needed to be filtered, and recommended. With that perspective, recommenders become something similar to data retrieval, where your query is something like your taste profile combined with the context, your mood and more. Netflix Senior Researcher Engineer

Mohammed Hossein Taghavi stated in his presentation at RecSys2016 that the ideal state of their recommender system was if you would "Turn on Netflix, and the absolute best contents for you would automatically start playing"¹²³.

But to do this we will need algorithms that can incorporate a larger model, which also includes much more knowledge of what the actual situation is for the current users. Is the user happy, with people, tired, etc.

In chapter 12 we talked about everything being an ensemble, which is also something that could support more inputs and more different models. Again, Deep learning is also seen as something that is expected to improve everything. What algorithms there is just around the corner is hard to say, maybe you will be the one to come up with the next big thing.

PRIVACY

With the social net, we will have more and more data about people and their connections. Even if collaborative filtering is good, because it connects user's behaviour, in the future we will also see that people would like to have recommendations based on close and trusted friends rather than a group of people that just happen to have the same taste as the active user. Trust based recommenders is already out there, and will this area will grow.

ARCHITECTURE

A recommender system will naturally be requested where there is too much choice. In the future, I guess, there will be choice everywhere, not only in the entertainment business like Netflix but all over, we will have doctors and lawyers not trusting to do their jobs without asking the recommenders first. Children will be getting recommendations on what to play with next to stimulate the child's brain optimally. With huge amounts of data, the recommender algorithms will start having problems and will simply be too time-consuming to get recommendations based on terabytes of data. We will, therefore, have to look for new types of algorithms which can handle to huge loads, or at least into optimizing the ones we have now.

As recommenders, will become something that runs everywhere we will also need recommenders which run on smaller devices, but with the rate that phones and other devices develop it is likely that it won't be a problem as they will have the same power as our servers have today. I could imagine that for example a topic model would be trained on a large amount of data, then the model can be used locally on your phone to do content based recommendations on your local book library.

¹²³ http://www.slideshare.net/MohammadHosseinTagha/balancing-discovery-and-continuation-in-recommendations?next_slideshow=1

SURPRISING RECOMMENDATIONS

One of the biggest problems of recommenders is that it is not very good at providing surprising recommendations, we need to figure out better ways to cut across categories and recommend things on a wider scale of the catalog. This will be one of the bigger problems to solve in the future. There are many proposals out there about how it could be done for example in *Novelty and Diversity Metrics for Recommender Systems: Choice, Discovery and Relevance*¹²⁴ Pablo Castells et al. lists different methods to measure these, as it is not straightforward even to do that.

14.4 Final Thoughts.

I should admit that I never actually read the last section in a machine learning nonfiction book, so I have spent some time researching what people tend to write here. And my conclusion is that they have all been sitting in the same situation as me.

As a parting thought about writing this book, I have met lots of people who have hinted to me that they had a great idea for a book and that if they bothered writing it, they would have written a much better book than what I have produced. Probably so. However, most of these have never actually started on the project. And for a good reason. To write a book is a humongous project, one that will teach you so much both on the subject that you are writing about but also on the art of writing a book, and will really test the strength of the bonds with your close ones. Regarding writing and recommender systems, I still have lots to learn. Regarding the bonds, I am happy to say that those were strong enough, even if I have at least a couple of years of making up for lost time with family and friends. If you do get the crazy idea of writing a technical book, then I can't recommend Manning enough, as they have made it a great experience.

Most would say that writing such book was a full-time job that wouldn't support you in any other way than giving you the gratifying feeling of the weight of the printouts. That may be. Writing this book, I have none the less learned a lot, I have met a lot of people, and gotten a lot of new contacts. I had fun and I hope you did too whilst reading and hopefully learned what you wanted. I will part with the wise words of Douglas Adams:

We are stuck with technology when what we really want is just stuff that works.

Douglas Adams (2002)

¹²⁴ <http://ir.ii.uam.es/rim3/publications/ddr11.pdf>