Recommender System

THESIS

Submitted in partial fulfillment of the requirements of
BITS F 423T/BITS F421T, Thesis

by

Aannesha Satpati
ID No- 2011A8PS324G

Under the supervision of

_____

Prof. Bharat Deshpande Head of Department of Computer Science

BITS, PILANI –K K BIRLA GOA CAMPUS

Date 08-05-2015

CERTIFICATE

This is to certify that the Thesis entitled, Recommender Systems

is submitted by Aannesha Satpati ID No . 2011A8PS324G

in partial fulfillment of the requirements of **BITS F 423T/BITS F421T** Thesis embodies the work done by him/ her under my supervision.

Signature of the supervisor

Prof. Bharat Deshpande
Head of Department of Computer Science

Date 08-05-2015

ABSTRACT

Recommender Systems have become an important technology in everyday applications. In information overload situations, such as social networks, e-commerce and streaming platforms personalized recommendations has proven to be a major source of enhanced functionality and revenue improvements. The development of recommendation algorithms and technologies has typically focused on maximizing the prediction accuracy of the user's interests. Here in this thesis we look at a brief introduction to recommender systems, and we look at famous recommender system cases and we have tried to build a simple recommender system in c language.

Content Page Number

Introduction

Recommender systems or **recommendation systems** (sometimes replacing "system" with a synonym such as platform or engine) are a subclass of information filtering system that seek to predict the 'rating' or 'preference' that user would give to an item.[1][2]

Recommender systems have become extremely common in recent years, and are applied in a variety of applications. The most popular ones are probably movies, music, news, books, research articles, search queries, social tags, and products in general. However, there are also recommender systems for experts, jokes, restaurants, financial services,[3] life insurance, persons (online dating), and Twitter followers.[4]

The analytics of the framework of a recommender system is as follows

- Doman
- Purpose
- Recommendation Context
- Opinion
- Personalization level
- Privacy and trustworthiness
- Interfaces
- Recommendation Algorithm

Let’s look at them one by one

Domain: This covers what is being recommended, there are different things that have to be recommended with a different purpose. The domains are broadly classified into two categories

- Commerce: Where the end goal is commercial for example products, matchmaking websites, and playlist
- One particular interesting property: This is a niece audience domain for example: movies and books

Purpose: The end goal of the recommender system has to be taken into account while building the recommender system. Usually the most common goals are as follows

- Sales
- Information
- Education
- Customers and products

Content: When the recommendation appears what the user is doing, as in what are the needs of the user. Depending on the environment the user is in and what he usually does in that particular environment the content has to be decided.

Options: This deals with the data and who’s opinions are considered for the data set. Generally it’s the experts opinions for example movie critics or it’s the general public.

Personalization Level: The level of personalization needed differs from system to system. The different level of personalization is as follows.

- Generic/non-personalized
- Demographic-target group

- Ephemeral-matches current activity
- Persistent- matches long time interest

Privacy and Trustworthiness: While making a recommender system one has to be careful about the privacy of the users, as personal information is revealed. Sometimes the user might want to have deniability of preference, that should be allowed by the system. Coming to trustworthiness sometimes a bias is build-in by the operator and that affects how the preferences work.

Interfaces: The output desired by the user has to be taken into account. The types of output as as follows

- Prediction
- Recommendation
- Filtering

Recommendation Algorithm: As the recommendation requirements differs so does the algorithm on a case to case basis.

Recommender systems typically produce a list of recommendations in one of two ways - through collaborative or content-based filtering.[5] Collaborative filtering approaches building a model from a user's past behavior (items previously purchased or selected and/or numerical ratings given to those items) as well as similar decisions made by other users; then use that model to predict items (or ratings for items) that the user may have an interest in.[6] Content-based filtering approaches utilize a series of discrete characteristics of an item in order to recommend additional items with similar properties.[7]

Collaborative filtering (CF) is a technique used by some recommender systems. Collaborative filtering has two senses, a narrow one and a more general one. In general, collaborative filtering is the process of filtering for information or patterns using techniques involving collaboration among multiple agents, viewpoints, data sources, etc. Applications of collaborative filtering typically involve very large data sets. Collaborative filtering methods have been applied to many different kinds of data including: sensing and monitoring data, such as in mineral exploration, environmental sensing over large areas or multiple sensors; financial data, such as financial service institutions that integrate many financial sources; or in electronic commerce and web applications where the focus is on user data, etc. The remainder of this discussion focuses on collaborative filtering for user data, although some of the methods and approaches may apply to the other major applications as well.

Another common approach when designing recommender systems is content-based filtering. Content-based filtering methods are based on a description of the item and a profile of the user’s preference. In a content-based recommender system, keywords are used to describe the items; beside, a user profile is built to indicate the type of item this user likes. In other words, these algorithms try to recommend items that are similar to those that a user liked in the past (or is examining in the present). In particular, various candidate items are compared with items previously rated by the user and the best-matching items are recommended. This approach has its roots in information retrieval and information filtering research

Non personalized recommender system

One of the very first websites to have incorporated recommender system was Zagat. Tim and Nina Zagat had created a restaurant guide couple of decades ago based on friends opinion on them. It shot to popularity as the website was interactive, later they sold it off to Google.

The recommender that they used was a non-personalized one. The formula used by then was a fairly simple one

Rating= {0,1,2,3}

*Score = round(mean(ratings)*10)*

CN traveler.com used a different approach where they show the percentage of people who rated a particular hotel.

Now the difference in both this approach is in Zagat each person's opinion has a lot more weight than in CN traveler.com

Now having seen the advantages of averages, they can be misleading as it lacks context and it makes the assumption that everyone will like the popular opinion.

Preference and Rating

To made a recommender system we need data from the user now that can be done in two ways

- Explicit
- Implicit

Lets look at them one at a time

Explicit: This is data collected through rating, reviews and voting. Common methods are the star rating which is hugely popular, thumbs and likes etc. The problem with this is are the ratings accurate an what if the user preference changes.

Implicit: Data collected from the user action like clicks on links and not clicks on links, purchases and what the user follow. Now this is not always accurate a user cannot click on a link because he/she didn't see it. This type of data collection can encroach on a person's privacy.

Hence recommenders mime the users and try to predict about their preference. Ratings are explicit and user behavior is implicit.

Personalized recommender system

Recommender systems are a particular type of personalisation that learn about a person's needs and then proactively identify and recommend information that meets those needs. Recommender systems are especially useful when they identify information a person was previously unaware of. Personalisation can be user-driven which involves a user directly invoking and supporting the personalisation process by providing explicit input. Examples of this include systems like MyYahoo! and MovieLens where the user explicitly initiates actions and provides example information in order to control the personalisation. Personalisation can also be completely automatic, where the system observes some user activity and identifies the input used to tailor some aspect of the system in a personalised way. These two examples of user-driven and automatic personalisation are at the extreme ends of the spectrum and many personalisation tools will have elements of both approaches. Personalisation systems have had great success in other areas. For example, in the area of targeted advertising we see tailored advertisements in the output pages from almost all web search engines. When using online retail systems such as Amazon.com we are given suggestions for additional complementary services and products. When using a WAP handheld device, the presentation arrangement of menu options is personalised and tailored for different users. Adaptive hypermedia systems demonstrate how personalisation can be used to assist a person in navigating through large, online courseware systems.

Here is a code to predict the rating of a user for a movie based on that user's rating history and other user's rating history in c

```
// RECOMENDATION SYSTEM FOR MOVIES (DATA SET HAS 943 USERS 1682 MOVIES)

#include<stdio.h>

#include<math.h>

#define NUSER 943

#define NMOV  1682

int matrix[NUSER+1][NMOV+1]/*={  -1,-1,-1,-1,-1,-1,

                                  -1, 2, 3,-1, 4,-1,

                                  -1, 2, 4, 1, 2,-1,

                                  -1,-1,-1, 2, 3, 2,

                                  -1, 4, 1, 5, 3, 2,

                                  -1, 1, 2, 4,-1, 5

                                 }*/ ;


float avg_user[NUSER+1];

float rms[NUSER+1];

float mod(float x)

{

        if(x<0)

        return -1*x ;
```

```
        else return x;
}
int generate_matrix()
{
        FILE* fp= fopen("data.txt","r");
        if(fp==NULL) {printf("File not found");exit(0);}
        int i,j,uid,mid,r;
        char ts[200];
        for(i=0;i<NUSER+1;i++)
        for(j=0;j<NMOV+1;j++)
                matrix[i][j]=-1;

        for(i=0;i<100000;i++)
        {
                fscanf(fp,"%d %d %d %s",&uid,&mid,&r,ts);
                 matrix[uid][mid]=r;
        }
}

int generate_average()
{
        int i,j,count=0,sum=0;
        float average;
        for(i=0;i<NUSER+1;i++)
         avg_user[i]=-1;

for(i=1;i<NUSER+1;i++)
 {
          for(j=1;j<NMOV+1;j++)
```

```
        {
                if(matrix[i][j]!=-1)
                {
                        sum=sum+matrix[i][j];
                        count++;
                }
        }
         average=(float)sum/count;
         avg_user[i]=average;
         sum=0;
        count=0;
 }
}



int generate_rms()
{
        int i,j;
        float msq=0,rmsq;
        for(i=1;i<NUSER+1;i++)
 {
  for(j=1;j<NMOV+1;j++)
  {
   if(matrix[i][j]!=-1)
        {
                msq=msq+ (matrix[i][j] - avg_user[i])*(matrix[i][j] - avg_user[i]);
        }
  }
```

```
  rmsq=sqrt(msq);
  rms[i]=rmsq;
  msq=0;
 }

}

float find_sim(int userid_1,int userid_2)
{

int i,j;
float num=0;
for(i=1;i<NMOV+1;i++)
 {
  if(matrix[userid_1][i]!=-1 && matrix[userid_2][i]!=-1)
            {
              num=num+ (   (matrix[userid_1][i] - avg_user[userid_1] )  *   ( matrix[userid_2][i]-avg_user[userid_2]
)  );
//            printf("%f is num,%d   %d  %f is av1 %f is
av2,%f\n",num,matrix[userid_1][i],matrix[userid_2][i],avg_user[userid_1],avg_user[userid_2],  (
(matrix[userid_1][i] - avg_user[userid_1] )  *   ( matrix[userid_2][i]-avg_user[userid_2] )  ));
            }
 }

float sim=num/(rms[userid_1]*rms[userid_2]) ;
return sim;

}
```

```
float predict(int userid,int movieid)
{
float predict_value,sim[NUSER+1];                //has similarity values of userid'th and i'th user
int i,j;


for(i=1;i<NUSER+1;i++)
 {
            sim[i]=find_sim(userid,i);
 }


float t2,sim_sum=0;

float num=0;

for(i=1;i<NUSER+1;i++)
 {    if(i==userid) continue;
      if(matrix[i][movieid]!=-1)
            num=num+ sim[i]*(matrix[i][movieid]-avg_user[i]);

 }

for(i=1;i<NUSER+1;i++)
 {
   sim_sum=sim_sum+mod(sim[i]);
 }
```

```
t2=num/sim_sum;

//printf("\n%f is sim_sum,%f is num ,%f is t2\n",sim_sum,num,t2);
predict_value=avg_user[userid]+t2;



return predict_value;
}



int main()
{

int i,j,k,userid=2,movieid=5;
generate_matrix();
generate_average();
generate_rms();

printf("Enter the user id to whom movies are to be recomended\n");

scanf("%d %d",&userid,&movieid);

if(matrix[userid][movieid]!=-1)
 {
  printf("Value already present:  %d\n",matrix[userid][movieid]);
  exit(0);
 }
```

```
float val=predict(userid,movieid);


printf("\n%f is value\n",val);



return 0;

}
```

Popular Recommender Systems

Online shoppers are accustomed to getting these personalized suggestions. Netflix suggests videos to watch. TiVo records programs on its own, just in case we're interested. And Pandora builds personalized music streams by predicting what we'll want to listen to.

All of these suggestions come from recommender systems. Driven by computer algorithms, recommenders help consumers by selecting products they will probably like and might buy based on their browsing, searches, purchases, and preferences. Designed to help retailers boost sales, recommenders are a huge and growing business. Meanwhile, the field of recommender system development has grown from a couple of dozen researchers in the mid-1990s to hundreds of researchers today—working for universities, the large online retailers, and dozens of other companies whose sole focus is on these types of systems.

Over the years, recommenders have evolved considerably. They started as relatively crude and often inaccurate predictors of behavior. But the systems improved quickly as more and different types of data about website users became available and they were able to apply innovative algorithms to that data. Today, recommenders are extremely sophisticated and specialized systems that often seem to know you better than you know yourself. And they're expanding beyond retail sites. Universities use them to steer students to courses. Cellphone companies rely on them to predict which users are in danger of switching to another provider. And conference organizers have tested them for assigning papers to peer reviewers.

Have you ever wondered what you look like to Amazon? Here is the cold, hard truth: You are a very long row of numbers in a very, very large table. This row describes everything you've looked at, everything you've clicked on, and everything you've purchased on the site; the rest of the table represents the millions of other Amazon shoppers. Your row changes

every time you enter the site, and it changes again with every action you take while you're there. That information in turn affects what you see on each page you visit and what e-mail and special offers you receive from the company.

Over the years, the developers of recommender systems have tried a variety of approaches to gather and parse all that data. These days, they've mostly settled on what is called the personalized collaborative recommender. That type of recommender is at the heart of Amazon, Netflix, Facebook's friend suggestions, and Last.fm, a popular music website based in the United Kingdom. They're "personalized" because they track each user's behavior—pages viewed, purchases, and ratings—to come up with recommendations; they aren't bringing up canned sets of suggestions. And they're "collaborative" because they treat two items as being related based on the fact that lots of other customers have purchased or stated a preference for those items, rather than by analyzing sets of product features or keywords.

Personalized collaborative recommenders, in some form or another, have been around since at least 1992. In addition to the GroupLens project, another early recommender was MIT's Ringo, which took lists of albums from users and suggested other music they might like.

GroupLens and Ringo both used a simple collaborative algorithm known as a "user-user" algorithm. This type of algorithm computes the "distance" between pairs of users based on how much they agree on items they have both rated. For instance, if Jim and Jane each give the movie *Tron* five stars, their distance is zero. If Jim then gives *Tron: Legacy* five stars, while Jane rates it three stars, their distance increases. Users whose tastes are relatively "near" each other according to these calculations are said to share a "neighborhood."

But the user-user approach doesn't work that well. For one thing, it's not always easy to form neighborhoods that make sense: Many pairs of users have only a few ratings in common or none at all, and in the case of movies, these few ratings in common tend to be of blockbusters that nearly everyone likes. Also, because the distance between users can change rapidly, user-user algorithms have to do most of their calculations on the spot, and that can take more time than someone clicking around a website is going to hang around.

So most recommenders today rely on an "item-item" algorithm, which calculates the distance between each pair of books or movies or what have you according to how closely users who have rated them agree. People who like books by Tom Clancy are likely to rate books by Clive Cussler highly, so books by Clancy and Cussler are in the same neighborhood. Distances between pairs of items, which may be based on the ratings of thousands or

millions of users, tend to be relatively stable over time, so recommenders can precompute distances and generate recommendations more quickly. Both Amazon and Netflix have said publicly that they use variants of an item-item algorithm, though they keep the details secret.

One problem with both user-user and item-item algorithms is the inconsistency of ratings. Users often do not rate the same item the same way if offered the chance to rate it again. Tastes change, moods change, memories fade. MIT conducted one study in the late 1990s that showed an average change of one point on a seven-point scale a year after a user's original rating. Researchers are trying different ways to incorporate such variables into their models; for example, some recommenders will ask users to rerate items when their original ratings seem out of sync with everything else the recommender knows about them.

But the user-user and item-item algorithms have a bigger problem than consistency: They're too rigid. That is, they can spot people who prefer the same item but then miss potential pairs who prefer very similar items. Let's say you're a fan of Monet's water lily paintings. Of the 250 or so paintings of water lilies that the French impressionist did, which is your favorite? Among a group of Monet fans, each person may like a different water lily painting best, but the basic algorithms might not recognize their shared taste for Monet.

About a decade ago, researchers figured out a way to factor in such sets of similar items—a process called dimensionality reduction. This method is much more computationally intensive than the user-user and item-item algorithms, so its adoption has been slower. But as computers have gotten faster and cheaper, it has been gaining ground.

To understand how dimensionality reduction works, let's consider your taste in food and how it compares with that of a million other people. You can represent those tastes in a huge matrix, where each person's taste makes up its own row and each of the thousands of columns is a different food. Your row might show that you gave grilled filet mignon five stars, braised short ribs four and a half stars, fried chicken wings two stars, cold tofu rolls one star, roasted portobello mushroom five stars, steamed edamame with sea salt four stars, and so forth.

A recommender using the matrix wouldn't really care about your particular rating of a particular food, however. Instead, it wants to understand your preferences in general terms, so that it can apply this knowledge to a wide variety of foods. For instance, given the above, the recommender might conclude that you like beef, salty things, and grilled dishes, dislike chicken and anything fried, are neutral on vegetables, and so on. The number of such taste attributes or dimensions would be much smaller than the number of possible foods—there

might be 50 or 100 dimensions in all. And by looking at those dimensions, a recommender could quickly determine whether you'd like a new food—say, salt-crusted prime rib—by comparing its dimensions (salty, beef, not chicken, not fried, not vegetable, not grilled) against your profile. This more general representation allows the recommender to spot users who prefer similar yet distinct items. And it substantially compresses the matrix, making the recommender more efficient.

It's a pretty cool solution. But how do you find those taste dimensions? Not by asking a chef. Instead, these systems use a mathematical technique called singular value decomposition to compute the dimensions. The technique involves factoring the original giant matrix into two "taste matrices"—one that includes all the users and the 100 taste dimensions and another that includes all the foods and the 100 taste dimensions—plus a third matrix that, when multiplied by either of the other two, re-creates the original matrix.

Unlike the food example above, the dimensions that get computed are neither describable nor intuitive; they are pure abstract values, and try as you might, you'll never identify one that represents, say, "salty." And that's okay, as long as those values ultimately yield accurate recommendations. The main drawback to this approach is that the time it takes to factor the matrix grows quickly with the number of customers and products—a matrix of 250 million customers and 10 million products would take 1 billion times as long to factor as a matrix of 250 000 customers and 10 000 products. And the process needs to be repeated frequently. The matrix starts to grow stale as soon as new ratings are received; at a company like Amazon, that happens every second. Fortunately, even a slightly stale matrix works reasonably well. And researchers have been devising new algorithms that provide good approximations to singular value decomposition with substantially faster calculation times.

By now, you have a basic idea of how an online retailer sizes you up and tries to match your tastes to those of others whenever you shop at its site. Recommenders have two other features that dramatically affect the recommendations you see: First, beyond figuring out how similar you are to other shoppers, the recommender has to figure out what you actually like. Second, the system operates according to a set of business rules that help ensure its recommendations are both helpful to you and profitable for the retailer.

For example, consider the recommender used for Amazon's online art store, which at last count had more than 9 million prints and posters for sale. Amazon's art store assesses your preferences in a few ways. It asks you to rate particular artworks on a five-star scale, and it also notes which paintings you enlarge, which you look at multiple times, which you place

on a wish list, and which you actually buy. It also tracks which paintings are on your screen at the time as well as others you look at during your session. The retailer uses the path you've traveled through its website—the pages you've viewed and items you've clicked on—to suggest complementary works, and it combines your purchase data with your ratings to build a profile of your long-term preferences.

Companies like Amazon collect an immense amount of data like this about their customers. Nearly any action taken while you are logged in is stored for future use. Thanks to browser cookies, companies can even maintain records on anonymous shoppers, eventually linking the data to a customer profile when the anonymous shopper creates an account or signs in. This explosion of data collection is not unique to online vendors—Walmart is famous for its extensive mining of cash register receipt data. But an online shop is much better positioned to view and record not just your purchases but what items you considered, looked at, and rejected. Throughout much of the world, all of this activity is fair game; only in Europe do data privacy laws restrict such practices to a degree.

Of course, regardless of the law, any customer will react badly if his or her data is used inappropriately. Amazon learned this lesson the hard way back in September 2000, when certain customers discovered they were being quoted higher prices because the website had identified them as regular customers, rather than as shoppers who had entered anonymously or from a comparison-shopping site. Amazon claimed this was just a random price test and the observed relationship to being a regular customer was coincidental, but it nevertheless stopped the practice.

The business rules around these systems are designed to prevent recommenders from making foolish suggestions and also to help online retailers maximize sales without losing your trust. At their most basic level, these systems avoid what's known as the supermarket paradox. For example, nearly everyone who walks into a supermarket likes bananas and will often buy some. So shouldn't the recommender simply recommend bananas to every customer? The answer is no, because it wouldn't help the customer, and it wouldn't increase banana sales. So a smart supermarket recommender will always include a rule to explicitly exclude recommending bananas.

That example may sound simplistic, but in one of our early experiences, our system kept recommending the Beatles' "White Album" to nearly every visitor. Statistically this was a great recommendation: The customers had never purchased this item from the e-commerce site, and most customers rated it highly. And yet the recommendation was useless. Everyone who was interested in the "White Album" already owned a copy.

Most recommender rules are more subtle, of course. When John recently searched for an action movie on Netflix, for instance, he wasn't offered *The Avengers*, because the blockbuster was not yet available for rental, and so the suggestion wouldn't have profited Netflix. Instead it steered him to *Iron Man 2*, which was available for streaming.

Other business rules prevent recommenders from suggesting loss leaders—products that sell below cost to draw people into the site—or conversely encourage them to recommend products that are overstocked. During our time at Net Perceptions, we worked with a client who did just that: He used his recommender system to identify—with considerable success—potential customers for his overstocked goods.

This kind of thing quickly gets tricky, however. A system that simply pushes high-margin products isn't going to earn the customers' trust. It's like going to a restaurant where the waiter steers you toward a particular fish dish. Is it really his favorite? Or did the chef urge the staff to push out the fish before its sell-by date?

To build trust, the more sophisticated recommender systems strive for some degree of transparency by giving customers an idea of why a particular item was recommended and letting them correct their profiles if they don't like the recommendations they're getting.

You can, for instance, delete information from your Amazon profile about things you purchased as gifts; after all, those don't reflect your tastes. You can also find out why certain products have been offered through the recommender. After Amazon selected Jonathan Franzen's novel *Freedom* for John, he clicked on the link labeled "Explain." He then got a brief explanation that certain books on John's Amazon wish list had triggered the recommendation. But as John hadn't read any of the wish list books, he discounted the *Freedom* suggestion. Explanations like these let users know how reliable a given recommendation is.

But profile adjustments and explanations often aren't enough to keep a system on track. Recently Amazon bombarded Joe with e-mails for large-screen HDTVs—as many as three a week for months. Besides sending him more e-mail on the topic than he could possibly want, the retailer didn't recognize that he'd already purchased a TV through his wife's account. What's more, the e-mails did not offer an obvious way for Joe to say, "Thanks, but I'm not interested." Eventually, Joe unsubscribed from certain Amazon e-mails; he doesn't miss the messages, and he has more time to actually watch that TV.

So how well do recommenders ultimately work? They certainly are increasing online sales; analyst Jack Aaronson of the Aaronson Group estimates that investments in recommenders

bring in returns of 10 to 30 percent, thanks to the increased sales they drive. And they still have a long way to go.

Right now the biggest challenge for those of us who study recommender systems is to figure out how best to judge the new approaches and algorithms. It's not as simple as benchmarking a microprocessor, because different recommenders have very different goals.

The easiest way to evaluate an algorithm is to look at the difference between its predictions and the actual ratings users give. For instance, if John gives the teen-romance novel *Twilight* one star, Amazon might note that it had predicted he would give it two stars, based on the ratings of other similar users, and so its recommender was off by a star. But sellers care much more about errors on highly rated items than errors on low-rated items, because the highly rated items are the ones users are more likely to buy; John is never going to purchase *Twilight*, so scoring this rating contributes little to understanding how well the recommender works.

Another common measure is the extent to which recommendations match actual purchases. This analysis can also be misleading, however, because it erroneously rewards the recommender for items users managed to find on their own—precisely the items they don't need recommendations for!

Given the shortcomings of these approaches, researchers have been working on new metrics that look not just at accuracy but also at other attributes, such as serendipity and diversity.

Serendipity rewards unusual recommendations, particularly those that are valuable to one user but not as valuable to other similar users. An algorithm tuned to serendipity would note that the "White Album" appears to be a good recommendation for nearly everyone and would therefore look for a recommendation that's less common—perhaps Joan Armatrading's *Love and Affection*. This less-popular recommendation wouldn't be as likely to hit its target, but when it did, it would be a much happier surprise to the user.

Looking at the diversity of a recommender's suggestions is also revealing. For instance, a user who loves Dick Francis mysteries might nevertheless be disappointed to get a list of recommendations all written by Dick Francis. A truly diverse list of recommendations could include books by different authors and in different genres, as well as movies, games, and other products.

Recommender systems research has all sorts of new ground to break, far beyond fine-tuning existing systems. Researchers today are considering to what extent a recommender should help users explore parts of a site's collection they haven't looked into—say, sending book buyers over to Amazon's clothing department rather than recommending safe items they may be more comfortable with. Going beyond the retail world, recommenders could help expose people to new ideas; even if we disagree with some of them, the overall effect might be positive in that it would help reduce the balkanization of society. Whether recommenders can do that without annoying us or making us distrustful remains to be seen.

How Hacker News ranking algorithm works

Hacker News is implemented in Arc, a Lisp dialect coded by Paul Graham. Hacker News is opensource and the code can be found at arclanguage.org. Digging through the news.arc code you can find the ranking algorithm which looks like this:

```
; Votes divided by the age in hours to the gravityth power.

; Would be interesting to scale gravity in a slider.


(= gravity* 1.8 timebase* 120 front-threshold* 1

  nourl-factor* .4 lightweight-factor* .3 )


(def frontpage-rank (s (o scorefn realscore) (o gravity gravity*))

 (* (/ (let base (- (scorefn s) 1)

       (if (> base 0) (expt base .8) base))

     (expt (/ (+ (item-age s) timebase*) 60) gravity))

   (if (no (in s!type 'story 'poll))  1

     (blank s!url)              nourl-factor*

     (lightweight s)             (min lightweight-factor*

                                 (contro-factor s))

                               (contro-factor s))))
```

In essence the ranking performed by Hacker News looks like this:

```
Score = (P-1) / (T+2)^G


where,

P = points of an item (and -1 is to negate submitters vote)

T = time since submission (in hours)

G = Gravity, defaults to 1.8 in news.arc
```

As you see the algorithm is rather trivial to implement. In the upcoming section we'll see how the algorithm behaves.

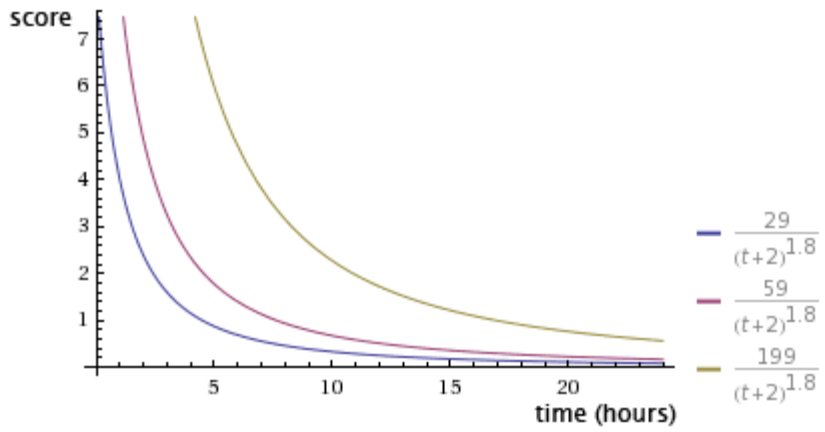
Effects of gravity (G) and time (T)

Gravity and time have a significant impact on the score of an item. Generally these things hold true:

- the score decreases as T increases, meaning that older items will get lower and lower scores
- the score decreases much faster for older items if gravity is increased

To see this visually we can plot the algorithm to Wolfram Alpha

How score is behaving over time



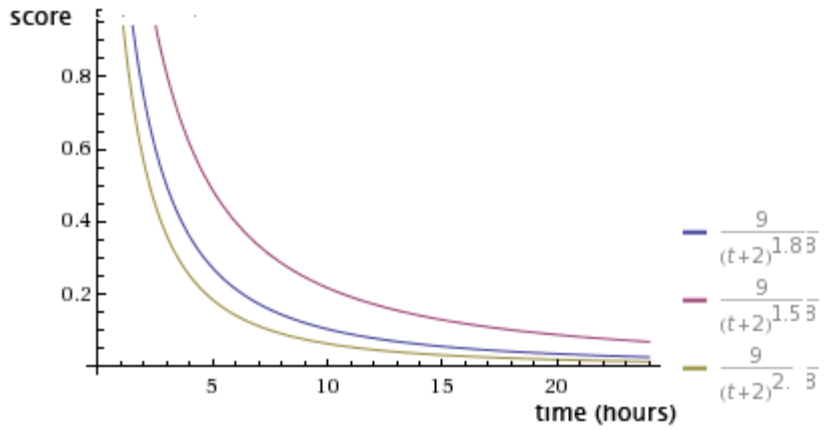
As you can see the score decreases a lot as time goes by, for example a 24 hour old item will have a very low score regardless of how many votes it got.

Plot query:

```
plot(
```

```
  (30 - 1) / (t + 2)^1.8,
  (60 - 1) / (t + 2)^1.8,
  (200 - 1) / (t + 2)^1.8
) where t=0..24
```

How gravity parameter behaves



As you can see by the graph the score decreases a lot faster the larger the gravity is.

Plotting query:

```
plot(
  (p - 1) / (t + 2)^1.8,
  (p - 1) / (t + 2)^0.5,
  (p - 1) / (t + 2)^2.0
) where t=0..24, p=10
```

Python implementation

As already stated it's rather simple to implementing the score function:

```
def calculate_score(votes, item_hour_age, gravity=1.8):
  return (votes - 1) / pow((item_hour_age+2), gravity)
```

Paul Graham has shared the updated HN ranking algorithm:

```
(= gravity* 1.8 timebase* 120 front-threshold* 1
   nourl-factor* .4 lightweight-factor* .17 gag-factor* .1)


(def frontpage-rank (s (o scorefn realscore) (o gravity gravity*))
  (* (/ (let base (- (scorefn s) 1)
          (if (> base 0) (expt base .8) base))
        (expt (/ (+ (item-age s) timebase*) 60) gravity))
     (if (no (in s!type 'story 'poll))  .8
         (blank s!url)                  nourl-factor*
         (mem 'bury s!keys)             .001
                                  (* (contro-factor s)
                                     (if (mem 'gag s!keys)
                                         gag-factor*
                                         (lightweight s)
                                         lightweight-factor*
                                         1)))))
```

How Reddit ranking algorithms work

Digging into the story ranking code: Reddit is open sourced and the code is freely available. Reddit is implemented in Python. Their sorting algorithms are implemented in Pyrex, which is a language to write Python C extensions. They have used Pyrex for speed reasons. I have rewritten their Pyrex implementation into pure Python since it's easier to read.The default story algorithm called the hot ranking is implemented like this:

```
#Rewritten code from /r2/r2/lib/db/_sorts.pyx


from datetime import datetime, timedelta

from math import log


epoch = datetime(1970, 1, 1)


def epoch_seconds(date):

  """Returns the number of seconds from the epoch to date."""

  td = date - epoch

  return td.days * 86400 + td.seconds + (float(td.microseconds) / 1000000)


def score(ups, downs):

  return ups - downs


def hot(ups, downs, date):

  """The hot formula. Should match the equivalent function in postgres."""

  s = score(ups, downs)

  order = log(max(abs(s), 1), 10)

  sign = 1 if s > 0 else -1 if s < 0 else 0

  seconds = epoch_seconds(date) - 1134028003
```

```
    return round(sign * order + seconds / 45000, 7)
```

In mathematical notation looks like this

Given the time the entry was posted A and the time of 7:46:43 a.m. December 8, 2005 B, we have t_s as their difference in seconds

$$t_s = A - B$$

and x as the difference between the number of up votes U and the number of down votes D

$$x = U - D$$

where $y \in \{-1, 0, 1\}$

$$y = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$$

and z as the maximal value, of the absolute value of x and 1

$$z = \begin{cases} |x| & \text{if } |x| \geq 1 \\ 1 & \text{if } |x| < 1 \end{cases}$$

we have the rating as a function $f(t_s, y, z)$

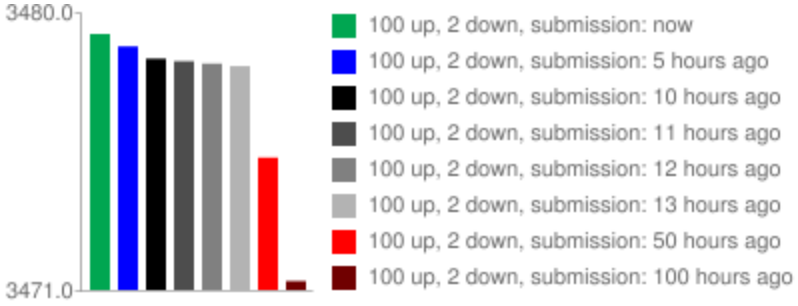
$$f(t_s, y, z) = \log_{10} z + \frac{yt_s}{45000}$$

Effects of submission time

Following things can be said about submission time related to story ranking:

- Submission time has a big impact on the ranking and the algorithm will rank newer stories higher than older
- The score won't decrease as time goes by, but newer stories will get a higher score than older. This is a different approach than the Hacker News's algorithm which decreases the score as time goes by

Here is a visualization of the score for a story that has same amount of up and down votes, but different submission time:



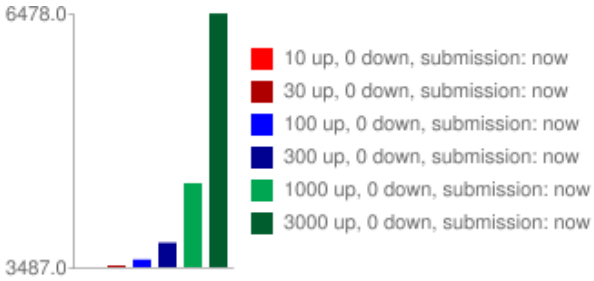
The logarithm scale

Reddit's hot ranking uses the logarithm function to weight the first votes higher than the rest. Generally this applies:

- The first 10 upvotes have the same weight as the next 100 upvotes which have the same weight as the next 1000 etc...

Here is a visualization:

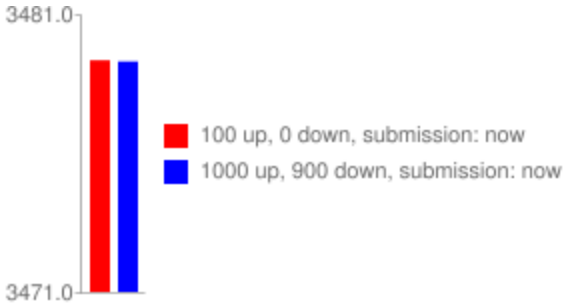
Without using the logarithm scale the score would look like this:



Effects of downvotes

Reddit is one of the few sites that has downvotes. As you can read in the code a story's "score" is defined to be: up_votes - down_votes

The meaning of this can be visualized like this:



This has a big impact for stories that get a lot of upvotes and downvotes (e.g. controversial stories) as they will get a lower ranking than stories that just get upvotes. This could explain why kittens (and other non-controversial stories) rank so high

How Reddit's comment ranking works

Digging into the comment ranking code

The confidence sort algorithm is implemented in _sorts.pyx

```
#Rewritten code from /r2/r2/lib/db/_sorts.pyx


from math import sqrt


def _confidence(ups, downs):
    n = ups + downs


    if n == 0:
        return 0


    z = 1.0 #1.0 = 85%, 1.6 = 95%
    phat = float(ups) / n
    return sqrt(phat+z*z/(2*n)-z*((phat*(1-phat)+z*z/(4*n))/n))/(1+z*z/n)


def confidence(ups, downs):
    if ups + downs == 0:
        return 0
    else:
        return _confidence(ups, downs)
```

The confidence sort uses Wilson score interval and the mathematical notation looks like this:

$$\frac{\hat{p} + \frac{1}{2n}z^2_{1-\alpha/2} \pm z_{1-\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{n} + \frac{z^2_{1-\alpha/2}}{4n^2}}}{1 + \frac{1}{n}z^2_{1-\alpha/2}}$$

In the above formula the parameters are defined in a following way:

- p is the observed fraction of positive ratings
- n is the total number of ratings

- $z_{\alpha/2}$ is the (1-α/2) quantile of the standard normal distribution

Let's summarize the above in a following manner:

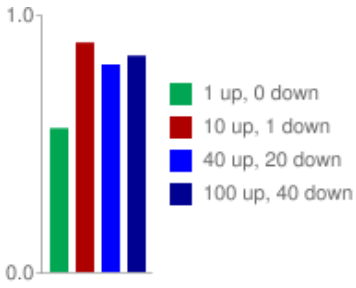
- The confidence sort treats the vote count as a statistical sampling of a hypothetical full vote by everyone
- The confidence sort gives a comment a provisional ranking that it is 85% sure it will get to
- The more votes, the closer the 85% confidence score gets to the actual score
- Wilson's interval has good properties for a small number of trials and/or an extreme probability

If a comment has one upvote and zero downvotes, it has a 100% upvote rate, but since there's not very much data, the system will keep it near the bottom. But if it has 10 upvotes and only 1 downvote, the system might have enough confidence to place it above something with 40 upvotes and 20 downvotes -- figuring that by the time it's also gotten 40 upvotes, it's almost certain it will have fewer than 20 downvotes. And the best part is that if it's wrong (which it is 15% of the time), it will quickly get more data, since the comment with less data is near the top.

Effects of submission time: there are none!

The great thing about the confidence sort is that submission time is irrelevant (much unlike the hot sortor Hacker News's ranking algorithm). Comments are ranked by confidence and by data sampling - - i.e. the more votes a comment gets the more accurate its score will become.

Visualization**Let's visualize the confidence sort and see how it ranks comments. We can use Randall's example:**



As you can see the confidence sort does not care about how many votes a comment have received, but about how many upvotes it has compared to the total number of votes and to the sampling size!

Application outside of ranking

Like Evan Miller notes Wilson's score interval has applications outside of ranking. He lists 3 examples:

- Detect spam/abuse: What percentage of people who see this item will mark it as spam?
- Create a "best of" list: What percentage of people who see this item will mark it as "best of"?
- Create a "Most emailed" list: What percentage of people who see this page will click "Email"?

To use it you only need two things:

- the total number of ratings/samplings
- the positive number of ratings/samplings

Given how powerful and simple this is, it's amazing that most sites today use the naive ways to rank their content. This includes billion dollar companies like Amazon.com, which define Average rating = (Positive ratings) / (Total ratings).

The Netflix Prize

One of the key events that energized research in recommender systems was the Netflix prize. From 2006 to 2009, Netflix sponsored a competition, offering a grand prize of $1,000,000 to the team that could take an offered dataset of over 100 million movie ratings and return recommendations that were 10% more accurate than those offered by the company's existing recommender system. This competition energized the search for new and more accurate algorithms. On 21 September 2009, the grand prize of US$1,000,000 was given to the BellKor's Pragmatic Chaos team using tiebreaking rules.

The most accurate algorithm in 2007 used an ensemble method of 107 different algorithmic approaches, blended into a single prediction:

Predictive accuracy is substantially improved when blending multiple predictors. *Our experience is that most efforts should be concentrated in deriving substantially different approaches, rather than refining a single technique.* Consequently, our solution is an ensemble of many methods.

Many benefits accrued to the web due to the Netflix project. Some teams have taken their technology and applied it to other markets, such as 4-Tell, Inc.'s Netflix project-derived solution for ecommerce websites.

A second contest was planned, but was ultimately canceled in response to an ongoing lawsuit and concerns from the Federal Trade Commission.

Conclusion

We have studied the basic aspects of recommender systems in thesis. We have looked at the aspects to be taken into account while making a recommender system. We have also looked at the different types of recommender systems, the pros and the cons of them. We have looked at the popular cases of recommender system world. We have developed a small predicting recommender system in C. This thesis just scratches the surface of the vast field of recommender system.

References

1. Francesco Ricci and Lior Rokach and Bracha Shapira, Introduction to Recommender Systems Handbook, Recommender Systems Handbook, Springer, 2011, pp. 1-35
2. How Computers Know What We Want — Before We Do
3. Alexander Felfernig, Klaus Isak, Kalman Szabo, Peter Zachar, The VITA Financial Services Sales Support Environment, in AAAI/IAAI 2007, pp. 1692-1699, Vancouver, Canada, 2007.
4. Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Bosagh Zadeh WTF:The who-to-follow system at Twitter, Proceedings of the 22nd international conference on World Wide Web
5. Hosein Jafarkarimi; A.T.H. Sim and R. Saadatdoost A Naïve Recommendation Model for Large Databases, International Journal of Information and Education Technology, June 2012
6. **Jump up^** Prem Melville and Vikas Sindhwani, Recommender Systems, Encyclopedia of Machine Learning, 2010.
7. **Jump up^** R. J. Mooney and L. Roy (1999). *Content-based book recommendation using learning for text categorization*. In Workshop Recom. Sys.: Algo. and Evaluation.

Bibliography

http://spectrum.ieee.org/computing/software/deconstructing-recommender-systems

http://amix.dk/blog/post/19574

http://amix.dk/blog/post/19588

http://en.wikipedia.org/wiki/Recommender_system

https://www.coursera.org/learn/recommender-systems