Deep Dive Into Movie Recommendation Systems

Cyrus Tabatabai-Yazdi
UCLA
ctabatab@cs.ucla.edu

Abstract

Recommendation systems are everywhere. Whether you are shopping online, reading news, or browsing Netflix, behind-the-scenes a recommendation system is trying to predict your next move before you even know it. One of the most popular techniques used for recommendations is collaborative filtering. Collaborative filtering comes in two flavors: memory-based and model-based. Memory-based is the traditional approach to collaborative filtering where past behavior is used to find similar users and items to make recommendations. Model-based approaches, made famous by the Netflix prize of 2006, depend on creating a model through machine learning or data mining algorithms for making recommendations. Both approaches have their benefits and drawbacks, and in this project, both types as well as content-based recommendations will be examined in detail. With the amount of data available on the Internet, finding a good recommendation is like finding a needle in the haystack, and the goal of this project is to understand the different approaches used and their effectiveness by creating a movie recommendation engine and framework using the MovieLens dataset.

Introduction

Long gone are the days where customers are limited to buying whatever is available in a store. With the help of the Internet, the number of potential items a customer can buy nowadays is infinite. Next time you do an Amazon search for an item you want, notice the vast amount of results that turn up. How can an online shopper filter out the massive amount of information on the web to find a useful item or even discover something new? This information overload necessitates a process to filter the information for the hidden gems. Although search engines do provide some of this, search engines today are mainly focused on matching results to keywords rather than personalizing to what users might find interesting or useful. Thus, many online shops and services have spent years devising algorithms that can predict a person's taste and preferences for making recommendations of what to buy or do next. For many of these online businesses, their end game is to convert a recommendation to a sale or retention. Billions of dollars have resulted from these recommendations while also helping people find things they need. Thus, it should come to no surprise that many interactions one might do online might be related to a recommendation. When you are browsing the news, the next article you end up reading could have been suggested. The next video shown automatically after a video finishes on YouTube is most likely a recommendation. The beauty of modern recommendation systems is that they are all data-driven by finding relationships between users and items based on past actions with no human curation involved. From the time when Amazon used recommender systems to filter books from its vast catalogue to today where it is the norm to have suggestions on what to read, watch, or buy next, recommendation systems have significantly evolved and the rest of this report will demystify the magic behind these systems by building a movie recommendation engine using the MovieLens dataset.

Brief History of Recommender Systems

Before diving into the technical details, it is worth going over the history of recommender systems as it provides a foundation for understanding the reasoning behind systems today. One of the earliest recommender systems was the UseNet platform by GroupLens which recommended news stories to

users based on what other users were reading [6]. GroupLens also created MovieLens in 1997, a platform where movie lovers could discover new movies based on the collaborative ratings of all users in the system. The MovieLens dataset we used was created using the real ratings of users in MovieLens. Pandora also started the Music Genome Project in 2000 for learning the similarities between music genres, artists, and songs to tailor recommendations. It was also around this time that Netflix started offering movie rentals online, using its proprietary Cinematch recommendation system to recommend movie rentals to customers. In 2006, Netflix hosted a competition for the best collaborative filtering algorithm that could improve on their own rating prediction system. This competition brought recommender systems to the spotlight, providing an avenue for the improvement and creation of new approaches to recommendations systems that could be applied across all domains. The winner of the competition used a model-based approach, a type of collaborative filtering where a model is learned through machine learning for making predictions and recommendations. Since then, many different techniques have come to the forefront, but the main ideas of traditional collaborative filtering are still very relevant [3].

Dataset

For this project, I chose to use MovieLens dataset made available by GroupLens. The dataset was collected from MovieLens website over different periods of times. Movielens is a movie recommendation website where users can rate movies in order to have recommendations. There are different sizes of datasets available based on the number of ratings by anonymous users who joined MovieLens in 2000. I chose the MovieLens 1M dataset which is a stable benchmark consisting of 1 million ratings from 60000 users on 40000 movies released in 2003. The dataset is divided into several .csv files — users, ratings, movies and tags. The user table contains the user id, gender, occupation, age and zip code. The movie file stores information such as movie id, title and genre. The ratings file contains the user id, movie id, rating, and timestamp. The ratings are made on a 5-star scale(1-5) and each user in the dataset is guaranteed to have made at least 20 ratings.

Framework

To facilitate the creation and evaluation of algorithms for recommendation systems, I made a recommendation engine and framework that allows researchers to focus on the details of implementing novel algorithms without worrying about data preprocessing and writing testing code. The framework is built on top of Python's Surprise library. The Surprise library was designed to aid in the building and analyzing of recommender systems. It alleviates the pain of dataset handling, has some common algorithms like SVD and KNN built in, and makes it easy to implement new algorithms just by overriding a base class. At a high level, it works by predicting the ratings of every movie for every user based on the algorithm used. The framework I built to extend Surprise consists of five different files that provide different functionalities. The *processing.py* file contains the Processing class which processes and parses the data, returning a dataset object compatible with Surprise and mappings of movies to years and genres and popularity rankings. The *metrics.py* file contains functions for evaluating recommender systems. It uses Surprise's built-in RMSE and MAE functions while providing top-n evaluation metrics like novelty, diversity, and hit rate which is discussed in more detail in the evaluations section. Creating an algorithm compatible with the framework is very easy. As mentioned before, the Surprise library makes it easy to implement new algorithms by inheriting from the AlgoBase base class. The AlgoBase class defines functions named *fit()* and *test()* which are used to train and test each algorithm. Thus, any custom algorithm we write will have these two methods available and we can override the *fit* method based on the training requirements of the custom algorithm. The only extra requirement is to implement the *estimate*() function which takes in two arguments, a user id and a movie id, and predicts the rating the user would give to that movie. This is where the code on how the custom algorithm makes rating predictions would be written. The *algorithm.py* file wraps a custom algorithm by creating an Algorithm instance with the custom algorithm as an instance variable, and implementing an *evaluate()* function that calls all the evaluation metrics on that algorithm. As in any machine learning algorithm, partitioning of data into training

and test sets is necessary. The *data_generator.py* file does this through the *Data* class which slices and dices the data from the dataset in all the ways needed to evaluate it. It now becomes easy to see how all these files and classes fit together. First, a Data instance can be created for a data set and create the evaluation data that will be used later. Then, after writing our custom algorithms, they can be wrapped by an Algorithm instance. The Evaluation class in *evaluation.py* takes in a list of Algorithm instances and calls the *evaluate()* function on each of those instances to evaluate each of the algorithms on the evaluation data generated earlier, generating the results in a nice tabular format. This framework makes it easy to test and evaluate new recommendation algorithms as prototyping new ideas rapidly is often a key part of any machine learning pipeline. It also allows for the easy comparison of multiple algorithms.

Methods

Content-Based Filtering

The first and simplest approach to making movie recommendations is to use content-based filtering. Content-based filtering works by recommending items based solely on the attributes of those items rather than using aggregate user behavior data. It is a simple yet intuitive and effective approach. For example, if someone likes *Star Wars*, it is conceivable that person may enjoy *Star Trek* due to both being of the same genre(science-fiction). The MovieLens dataset associates a list of genres and the release year with each movie. The year can be considered important because if someone prefers older science fiction movies, it is reasonable to recommend science fiction movies close in year. The approach I took was to incorporate year and genre in order to make content-based recommendations. To find movies similar in genres, cosine similarity can be used which works by calculating the cosine of the angle between two vectors. As mentioned before, there are 18 possible genres for a movie to belong to. I encoded the genres a movie belongs to into a vector of length 18 which consists of 0's and 1's. A value of 1 in an entry in the vector means the movie belongs to that genre and a value of 0 means it does not belong to that genre. Thus, by encoding each movie as an 18-dimensional vector, cosine similarity can be used to find similar movies in this new genre vector space. The equation for cosine similarity in a multi-dimensional space is simple and shown below:

$$cos(\vec{a}, \vec{b}) = \frac{\vec{a}.\vec{b}}{\|\vec{a}\|^2 * \|\vec{b}\|^2}$$

Incorporating the year into similarities requires a bit of art. We must consider how far apart two movies would have to be for their release date alone to signify they are significantly different. I chose a decade as a reasonable starting point but this is a hyperparameter that can be tuned. The absolute value of the difference in the release years of two movies would be passed to an exponential decay function that smoothly scales to the range 0 to 1. The exponential decay function I used is shown below. A year difference of 0 would lead to a value of 1 while a year difference of 10 would be very small and further increases in the year difference would be almost 0. The reason I chose this function may seem arbitrary, but this again comes down to the art of recommender systems rather than the science. In the real world, many variations of this function would be tested on real people to see what really produces the best recommendations. The final similarity score between two mvoies will be weighted multiplication of the genre similarity and year similarity. The job is not finished though once we have found similarity scores between movies based on either genre, release year, or a combination of both. We do not only want to have similarities between movies but actual predicted ratings that can be turned into recommendations and the framework I developed is designed to predict a rating for a given user for a given movie (the *estimate()* function discussed in the Framework section). A simple way to do this is a technique called k nearest neighbors or KNN. To find the rating a user might give a particular movie, we first find the *k* nearest neighbors to that movie where those nearest neighbors are movies that the user has rated and which have the highest similarity scores to the movie. The weighted average of the similarity scores of the movies to the movie we want

to score can be taken for a predicted rating, weighing them by the rating the user gave them.

A state-of-the-art approach to content-based filtering I looked at and implemented is based on the paper "Using Visual Features and Latent Factors for Movie Recommendations" []. The idea behind this method is to use "mise en scene" data. Mise on scene refers to the properties of the scenes in a movie or movie trailer. This means the properties from the movie itself will be extracted that can be quantified and analyzed to see if better recommendations are possible. The authors diligently created a dataset for movies in the MovieLens dataset that maps each movie to a few attributes. These attributes include average shot length, color variance, how much motion is in each scene, how scenes are lit, and the variance in lighting. In principle, these attributes should give a feel for the pacing and mood of the film. Are using these features more useful than using genres or years? The evaluation section shows that it did a key theme in recommendation systems that needs to be repeated is that offline evaluations should not have the final say.

The advantages of content-based filtering is that the recommendations it provides are intuitive and make sense. If a person likes fantasy movies, it is fair to assume they will like watching new fantasy movies they have not seen it. However, a disadvantage is overspecialization in that the recommendations can be pigeonholed into a few genres, so the diversity of the recommendations may be low. However, content-based filtering is a simple yet effective way to provide recommendations and is often used in conjunction in hybrid models with the more advanced approaches discussed next.

Neighborhood-Based Collaborative Filtering

Memory-based collaborative filtering, also known as neighborhood-based collaborative filtering is a classical approach to making recommendations by leveraging the aggregate behavior of users. At a very high level, it means finding other people like you and recommending things they liked or finding other items similar to items you have liked. Essentially, it takes cues from people who are like you – your "neighborhood" – and recommending items based on things they liked that you have not seen yet. It is called "collaborative" filtering because the recommendations are based on peoples' collaborative behavior (the ratings people have given to movies). The canonical architecture for collaborative filtering is shown below as is used by Amazon for item-item collaborative filtering []. First, the ratings data (a matrix consisting of users as rows, movies as columns, and ratings as entries) is used to generate candidates for movies to recommend by looking up other movies similar to the movies each user liked. Next, each of these candidates is scored and ranked and then movies already seen are filtered out to produce a final top-N recommendation list that can be displayed to the user. The heart of this approach is the database of item similarities or user similarities. Thus, there needs to be a way to quantify the similarity between users or items which will be discussed next.

Similarity Metrics

One of the big challenges in computing this similarity is the sparsity of the ratings data. Most movies have not been watched by a given individual, and conversely, most people have not seen a given movie. This makes collaborative filtering hard to work well unless there is a lot of user behavior data to work with. There is no meaningful similarity between two people when they have no movies in common, or between two movies when they have no people in common. The 2D matrix of ratings by users for movies that we start with will be mostly empty in practice. Companies like Netflix and Amazon have millions of users and so have enough data to generate meaningful relations despite sparsity. For this project, the 1 million ratings is not enough to provide excellent similarity data, but suffices for this project. It is important to note that the quantity and quality of data being worked on is oftentimes more important than the algorithms used, and this applies across all scientific disciplines. Assuming excellent and voluminous data, there are a few similarity measures that can be used to calculate similarity between users or items. Cosine similarity, as used in content-based filtering, is a common approach as well as each row of the user-movie ratings matrix can be treated as a vector. The dimensions in this case will be "did this user like this movie" or "was this movie liked by this user" instead of genre membership. A variation of cosine similarity is the adjusted cosine similarity

shown below. It is most applicable for measuring the similarity between users as it is based on the idea that different people might have different baselines that they are working from – what Alice considers a 3-star movie may be different from what Bob considers a 3-start movie. Also, Alice might be hesitant to rate movies 5 stars unless they are truly amazing, while Bob is generous and rates everything 5 stars unless he really did not like the movie. Thus, the adjusted cosine similarity metric tries to normalize these differences by measuring similarity based on the difference between users' ratings for a movie and the users' average ratings for all movies. The difference from cosine similarity is that we are looking at the variance from the mean of each user's ratings rather than the raw rating itself as shown below:

$$s(u,v) = \frac{\sum_{i \in \mathcal{I}_{uv}} (r_{ui} - \bar{r}_i)(r_{vi} - \bar{r}_i)}{\sqrt{\sum_{i \in \mathcal{I}_u} (r_{ui} - \bar{r}_i)^2 \sum_{i \in \mathcal{I}_v} (r_{vi} - \bar{r}_i)^2}}$$

This metric sounds good on paper, but data sparsity can present a lot of challenges as a meaningful average rating can occur only if individuals have rated many movies. The dataset I am working with guarantees all users have rated at least 20 movies, but in the real world, there may be many new users who have only rated one thing and that data will be wasted with this metric as the difference will be 0 regardless of what the new user rated that one movie. Another common metric is the Pearson similarity. It is a slight twist on the adjusted cosine as instead of looking at the difference between ratings and a user's average rating, we look at the difference between the ratings from all users for a given movie. This means we are no longer trying to account for an individual's personal definitions of specific rating scores, and in a real-world situation with sparse data, that can be seen as a good thing. At a high level, Pearson similarity can be seen as measuring the similarity between people by how much they diverge from the average person's behavior. For example, imagine a film that most people love, like *Harry Potter*. People who hate *Harry Potter* will have a strong Pearson similarity score because they share a sentiment that is not mainstream. The equations for the Pearson similarity between users and items are shown below.

$$s(u,v) = \frac{\sum_{i \in \mathcal{I}_{uv}} (r_{ui} - \bar{r}_i)(r_{vi} - \bar{r}_i)}{\sqrt{\sum_{i \in \mathcal{I}_u} (r_{ui} - \bar{r}_i)^2 \sum_{i \in \mathcal{I}_v} (r_{vi} - \bar{r}_i)^2}}$$

$$s(i,j) = \frac{\sum_{u \in \mathcal{U}_{ij}} (r_{ui} - \bar{r}_i)(r_{ui} - \bar{r}_j)}{\sqrt{\sum_{u \in \mathcal{U}_{ij}} (r_{ui} - \bar{r}_i)^2 \sum_{u \in \mathcal{U}_{ij}} (r_{ui} - \bar{r}_j)^2}}$$

Another way to measure similarity is to use the mean squared distance similarity metric. It works by looking at all the items two users have in common or all the users two items have in common and computes the mean squared difference between how each user rated each common item or how each item was rated by each common user. The equation for user-based collaborative filtering is shown below.

$$s(u,v) = \frac{\sum_{i \in \mathcal{I}_{uv}} (r_{ui} - r_{vi})^2}{|\mathcal{I}_{uv}|}$$

The MSD formulation for item-based collaborative filtering follows the same template. It is an easier metric to wrap our heads around because it does not involve angles in multi-dimensional space and is similar to loss functions used in other machine learning algorithms. At the end, the inverse of the MSD is taken because we want the similarity of two items or users, not how different they are. A higher MSD score means less similar while a lower MSD score means more similar. I will briefly mention a final similarity metric called the Jaccard similarity. The Jaccard similarity works by counting up all the movies that both users have indicated some sort of interest in and divide that by total number of movies that either user has indicated interest in. The key here is that implicit ratings, like the fact that somebody watched something or bought something, are prevalent and the Jaccard similarity is a reasonable

choice that is fast to compute in these scenarios. Whichever similarity matrix is chosen is more of an art than science and can be validated through online live A/B testing. In practice, cosine similarity and Pearson similarity have been used extensively, including by Amazon[11].

With the similarities between users or items calculated, the hardest part of neighborhood-based collaborative filtering is done. There are two types of neighborhood-based collaborative filtering: user-based and item-based. The only difference is whether we are looking at user-user similarities or item-item similarities.

User-Based collaborative filtering

In user-based collaborative filtering, the idea is to find other users similar to yourself based on their ratings history, and then recommend movies they enjoyed that you have not seen yet. For example, if Alice and Bob both like *Harry Potter* and Bob has not seen *Lord of the Rings* while Alice enjoyed it, it is conceivable that Bob will enjoy it as well. The input is the ratings matrix where the users are the rows and the movies are the columns and each entry is the rating given by a user to a movie. As noted, many times before, this matrix will be sparse. An example ratings matrix is shown in the following table:

	Lord of the Rings	*Harry Potter*	*Star Wars*	*Game of Thrones*	*Incredibles*
Alice		5	5	5	
Bob	4	5			
Ted					5

Each user can be described by a vector that would be the n-dimensional where n is the number of movies that exist in the system. In our case, it is five for simplicity. The vectors that represent Alice, Bob, and Ted are <0,5,5,5,0>, <4,5,0,0,0>, and <0,0,0,0,5>. respectively where 0 represents a missing value. Any similarity metric can be used but let us assume we are using cosine similarity for illustration purposes. Obviously, every user is 100% similar to themselves if they have rated at least one movie. Ted and Bob have no movies in common that they both rated, so they get a similarity score of 0. The same applies to Alice and Ted. It is important to recognize the matrix is symmetric which can be exploited to save computing half the values in the similarity matrix. Alice and Bob have one movie in common that they have both rated: *Harry Potter.* They both gave it 5 stars so the similarity score between Alice and Bob will be 1.0. A caveat of cosine similarity is that if two users have only one movie in common, they will have 100% similarity regardless of ratings. This means that if Alice had rated *Harry Potter* 1 star instead, the similarity score would still be 1.0. This is once again a side effect of sparsity so in the real-world, it is important to enforce a minimum threshold on how many movies users have in common before considering calculating the similarity between them. The user-user similarity matrix for the toy example above is shown below:

	Alice	Bob	Ted
Alice	1.0	1.0	0.0
Bob	1.0	1.0	0.0
Ted	0.0	0.0	1.0

Say we want to generate recommendations for Alice now. We would look at the similarity scores between Alice and all other users in the system, sort the list in descending order, and pick off the top N neighbors who have the highest similarity scores to Alice. We then pull all of the movies users in the top N neighbors list have rated and consider them as recommendation candidates. Next, we need to figure out which of these recommendation candidates are the best ones to present to Alice, so we need to score them somehow. There are different ways to do this, but it is reasonable to consider the rating assigned to each candidate movie by the top N similar users as we want to recommend things similar users loved, not things that similar users hated. It is also prudent to consider the similarity of the user who generated these ratings into consideration. Thus, the final scoring function is shown below:

$$\hat{r}_{ui} = \frac{\sum_{v \in \mathcal{N}_i(u)} w_{uv} r_{vi}}{\sum_{v \in \mathcal{N}_i(u)} |w_{uv}|}$$

Here, $N_i(u)$ is the set of top-N similar users who have rated item *i*. r_{vi} is the rating given to item *i* by user *v* and w_{uv} is the similarity score between user *u* and

user *v*. Once the predicted ratings user *u* would give items with no ratings are calculated, the top-N highest ratings can be returned for recommendations with N dependent on the context and use case of the application.

Item-Based Collaborative Filtering

Another way to do nearest-neighbor collaborative filtering is by flipping the problem on its head – instead of looking for other people similar to you and recommending movies they liked, we look at the movies you liked, and recommend movies similar to those movies. This is known as item-based collaborative filtering which has been used to great success at Amazon. Looking at the similarities between movies has a few advantages over looking at the similarities between people. One reason is that movies tend to be of a more permanent nature than people. A *Harry Potter* movie will always be a *Harry Potter* movie, but an individual's tastes may change over time. A *Harry Potter* movie will always be similar to other coming-of-age fantasy movies, but a person might be interested in documentaries for a short period of time before losing interest and moving on to another genre or other types of movies. This means the item similarity matrix can be computed less frequently as it does not change as quickly. Another key advantage is that there are typically fewer movies than people in a system. There are more people using a service than movies to recommend which is certainly true of Netflix and Amazon. This means that the similarity matrix of size #movies by #movies will be significantly smaller than the similarity matrix for user-based collaborative filtering which would be of size #users by #users. Thus, it will be faster and more efficient to compute and store and with large scale systems like Netflix and Amazon, computational efficiency is very important. Not only does it require fewer resources, but the similarity matrix can be regenerated more often, making the recommendation system more responsive when new movies are introduced. Lastly, using item similarities creates a better experience for new users, mitigating the effects of the cold-start problem. When a new user comes to use a service and they have indicated an interest in a movie, movies similar to that movie can be recommended. With user-based collaborative filtering, no recommendations for that new user would exist until they make it into the next build of the user-user similarity matrix, which, as mentioned before, is expensive to compute.

Item-based collaborative filtering is very similar to user-based collaborative filtering, but with a few minor differences. Instead of starting with a ratings matrix that has users as rows and movies as columns, we start with the movies as rows and users as columns. Thus, the ratings matrix from the user-based example would become like this:

	Alice	*Bob*	*Ted*
Lord of the Rings		4	
Harry Potter	5	5	
Star Wars	5		
Game of Thrones	5		
Incredibles			5

Then, we can once again calculate the similarity between items using any of the similarity metrics discussed before by thinking of the movies existing in a vector space where ever user is a dimension. The result would be a #items by #items matrix. This similarity matrix could then be consulted in the same way as in user-based. For example, if a new user, say Daniel, comes in and likes *Star Wars.* We can consult our item-item similarity matrix to look up movies similar to *Star Wars* based on the ratings of other uses who watched *Star Wars* in the past. If we want to generate how a user would rate a movie they have not seen, we would find the *N* most similar movies to that movie that have also been rated by this user and compute a weighted average through the equation shown below to predict the rating user *u* would give movie *i*:

$$\hat{r}_{ui} = \frac{\sum_{j \in \mathcal{N}_u(i)} w_{ij} r_{ju}}{\sum_{v \in \mathcal{N}_u(i)} |w_{ij}|}$$

In the equation above, w_{ij} represents the similarity between item *i* and item *j*. $N_u(i)$ is the set of the most similar *N* items that that user *u* has also rated . r_{ju} is the rating given to item *j* by user *u*. Again, the top-N

movies can be shown the user u based on the highest predicted ratings for movies they have not seen.

Both user-user collaborative filtering and item-item collaborative filtering follow the same basic framework, summarized below.

```
Input: User-Item Rating matrix R
Output: Recommendation lists of size l
Const l: Number of items to recommended to user u
Const v: Maximum number of users in N(u), the neighbours of user u
For each user u Do
    Set N(u) to the v users most similar to user u
    For each item i that user u has not rated Do
        Combine ratings given to item i by neighbours N_i(u)
    End
    Recommend to user u the top-l items with the highest predicted rating r̂_ui
End
```

Figure 3: Collaborative Filtering Pseudocode.

Neighborhood-based collaborative filtering is a very intuitive and simple approach to making recommendations and has been used to great success in the past. The main benefits are that the recommendations are intuitive and explainable and, once the similarity matrix is computed, generating recommendations is quick. The reason the recommendations are intuitive is because the way neighborhood-based collaborative filtering works is to recommend what users similar to you liked or items similar to what you liked. Thus, it is easy to explain to a user why the recommendations are valid. Also, the only computationally heavy part of this method is generating the similarity matrix. Once the similarity matrix has been generated, finding recommendations becomes nothing more than a lookup in the matrix which is very efficient. However, as mentioned before, one of the main challenges to neighborhood-based collaborative filtering is data sparsity and scalability. The sparsity issue is valid as sparse data makes similarity scores less meaningful but the scalability issue is not as sound as item-based collaborative filtering is more scalable and using Apache Spark can allow for distributed computing to compute the similarity matrices. Regardless, most new advances in recommendation systems and collaborative filtering have moved away from nearest-neighborhood approaches to using models.

Model-Based Collaborative Filtering

The high-level idea behind model-based collaborative filtering is, instead of finding movies or users similar to each other, data science and machine learning techniques are applied to extract predictions for missing ratings (movies a user has not rated). Machine learning is all about training models to make predictions on unseen data so the problem of making recommendations can be treated the same way- models will be trained on the ratings data and those models will be used to predict the ratings of new movies by users.

Matrix Factorization

Matrix factorization for recommendations came to the forefront during the 2006 Netflix challenge and it has firmly remained in the picture and is still widely used in industry. There are a variety of techniques that fall under the category of matrix factorization. Regardless, the general idea is to describe users and movies as combinations of different amounts of latent features. For example, Alice can be defined as being an 80% science-fiction fan and 20% a thriller fan. Thus, we would know to recommend movies that are a blend of about 80% science-fiction and 20% thriller. Obviously, the numbers would not be percentages, but the principle remains the same. These latent features such as a user being a science fiction fan are found by the magic of matrix factorization.

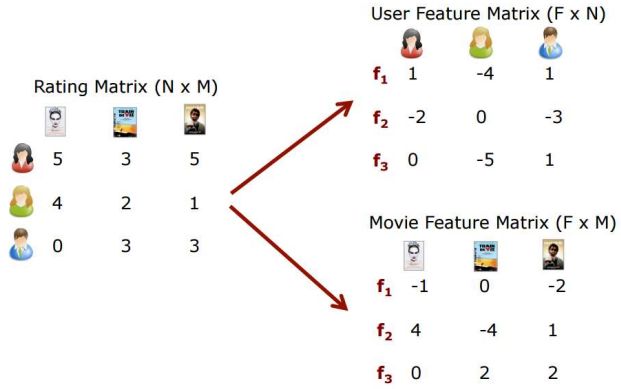

Figure 1: Latent Factors for Users and Movies [8]

It is important to think backwards when looking at how matrix factorization works. What is the goal

when we are trying to predict ratings for users? All it is trying to fill in the values in a sparse matrix. The ratings matrix exists with users as rows and movies as columns and most of the entries in the matrix are unknown. The challenge matrix factorization and other model-based approaches try to overcome is how to fill in those empty entries with predicted ratings. One approach to this challenge is principal component analysis (PCA).

PCA can be described as dimensionality reduction method. In other words, we take data that exists in many dimensions, like all the movies a user could watch, and transform it into a smaller set of dimensions that can accurately describe a movie, such as its genres. The goal of PCA is to find the "principal components" that best describe our data which are given by eigenvectors. The canonical example to example PCA is to look at the Iris dataset, which describes the length and width of petals and sepals for different Iris flowers. These principal components are useful because they represent the variance of the data and can find latent features inherent in the data. In other words, PCA can be viewed as a feature extraction tool as well. Running PCA on the 4-dimensional Iris dataset results in the finding of two dimensions that best represent the data. It should be noted that PCA returns as many dimensions as in the original data, but the least important dimensions can be discarded. What these dimensions mean exactly is vague as what does the X axis mean and what does the Y axis mean? All we know is they represent latent features that PCA extracted from data, but only human analysis can put some sort of label on those features. Running PCA on the ratings matrix where every dimension is a movie can boil this high-dimensional space into a much smaller number of dimensions that best describes the variance in the data. Often, the dimensions it finds correspond to features humans have learned to movies as well such as how action-y a movie is or how funny a movie is. Whatever it is about movies that causes people to rate them differently can be found by PCA through latent features extracted from the data. As an illustration, consider the below sample ratings matrix for three users and five movies. Obviously, in the real-world, the matrix will be much bigger and there will be many missing entries, but it suffices as an illustrative example:

	Lord of the Rings	*Harry Potter*	*Star Wars*	*Game of Thrones*	*Incredibles*
Alice	3	5	5	5	2
Bob	4	5	1	2	5
Ted	2	3	4	5	5

PCA can find a matrix U with users as rows and the columns being the latent features. PCA will not know what these latent features are exactly but let us suppose they end of being measures of a person's interest in action, science-fiction, and fantasy genres.

	Action	*Sci-Fi*	*Fantasy*
Alice	0.3	0.2	0.5
Bob	0.1	0.3	0.6
Ted	0.4	0.4	0.2

Thus, we can think of Alice as being defined as 30% action, 20% science-fiction, and 50% fantasy. Each column (latent feature), is a description of the users that make up that feature. For example, action can be described as 30% of Alice, 10% of Bob, and 40% of Ted. Just as we can run PCA on our user-movie ratings data, we can flip things around to find the latent features of movies. If we make our ratings matrix be composed of movies as rows and users as columns(transpose or original matrix), running PCA will once again identify the latent features and describe each individual movie as some combination of them. Each column would be a description of some typical movies that exhibits some latent features. This matrix can be called M.

	Action	*Sci-Fi*	*Fantasy*
Lord of the Rings	0.1	0.4	0.5
Harry Potter	0.3	0.2	0.5
Star Wars	0.3	0.6	0.1
Game of Thrones	0.6	0.1	0.3
Incredibles	0.5	0.2	0.3

Once these matrices have been found, the question is how do these matrices that describe typical users and typical movies help us predict missing ratings? The answer is that the typical movie matrix M and the typical user matrix U's transpose are both factors of the original ratings matrix we started with. This means that if we have M and U, we can reconstruct R. This R will have missing ratings in the real world

so if we have M and U, we can fill in all of those missing entries. This is why this method is called matrix factorization: our training data of ratings is described in terms of smaller matrices that are factors of the ratings we want to predict as shown below.

$$R = M\Sigma U^T$$

The Σ is a simple diagonal matrix that only serves to scale the values we end up with into the proper scale. This means that R can be reconstructed by multiplying all these factors at once, therefore generating the ratings for every combination of users and movies. It should also be noted that once these factors (M and U) are found, a rating can be predicted for a specific user and movie by taking the dot product of the associated row in M for the user, and the associated column in U for the item. If the equation above looks familiar, it is because it is nothing more than Singular Value Decomposition (SVD). SVD was widely used during the Netflix Prize amongst the leaders in the competition. All SVD is a way of computing M, Σ, and U-T all at once very efficiently. This means it is simply running PCA on both the users and movies and returning the matrices we need that are factors of the original ratings matrix. Thus, we get all three factors in one shot.

A question you might be asking yourself is how we can compute M and U-T using SVD or PCA when the original matrix R is missing most of its values due to sparsity as PCA and SVD can only be run on complete matrices. The original approach was to fill in the missing entries with some sort of reasonable default value such as mean values. However, there is a better way because every rating can be described as the dot product of some row in the matrix M and some column in the matrix U-T. For example, if we want to predict Alice's rating for Harry Potter, we can compute that predicted rating as the dot product of the Alice row in M and Harry Potter column in U-T. Assuming we have some known ratings for any given row and column in M and U-T. Thus, this problem can be treated as a minimization problem where we are tying to find the values of the missing entries that best minimizes the errors in the known ratings in R. Many machine learning techniques thrive on these optimization problems such as Stochastic Gradient descent and Alternating Least Squares. In the end, we are trying to learn the values in the matrices M and U rather than computing them directly as SVD would do. This means we are not doing "real" SVD as we are working with incomplete matrices. However, the algorithm that was used to win the Netflix Prize was inspired by SVD. In fact, the winning team called BellKor, used a combination of a specific variant of SVD++ and Restricted Boltzmann machines [11]. For this project, I did not implement matrix factorization techniques from scratch as the Surprise library has both SVD and SVD++ built in and both run very efficiently. My main focus on model-based approaches was to implement and analyze deep learning approaches to recommendations which we will look at next.

Deep Learning

We all know that deep learning is all the rage right now. It is hard to find a field in which deep learning has not been applied and it is not surprising there is a lot of research around applying deep learning to recommendation systems.Deep learning has been categorized as a one size fits all solution to many problems. Google itself has vowed to use deep learning on all its problems. However, just because a technology or technique has worked successfully for many applications does not always mean it is the right solution for every problem. A neural network can be trained to make predictions for ratings, but is it the right approach? Deep learning is good at recognizing patterns, but is making recommendations really a pattern recognition problem. In a way, recommendation systems can be seen as looking for patterns in terms of the collaborative behavior of many people. Since matrix factorization itself can be modeled as a neural network, it makes sense that deep learning could possibly work and all the hype is justified and companies like Amazon and Netflix have said they are migrating to incorporating deep-learning based recommendations in their recommendation engines. For example, Amazon has open sourced a system called Destiny(DSSTNE), which allows for large neural networks that works with sparse data to be run on clusters efficiently and they are personally using that system for their recommendations. Also, Tensorflow's computational graphs can always be run on a cluster and take advantage of a whole fleet of GPU's. Many approaches today using neural networks have been shown to out-perform SVD already, even if by small

margins. This shows the promise of deep learning and my goal was to look at two state-of-the-art architectures and implement them to test them against traditional approaches.

Restricted Boltzmann Machines

Restricted Boltzmann Machines have been around since 2007 [10] but it is still in heavy use today as the original paper is still commonly cited. In fact, Amazon has said they still incorporate RBMs in their recommendation system. One of the main things Netflix learned from the 2006 challenge was that matrix factorization and RBM's had the best performance in terms of RMSE. Combining matrix factorization with RBM's led to even better results, reducing the RMSE from 8.9 to 8.8. As recently as 2012, Netflix confirmed they were still using RBM's as part of their recommendation system, so it is clearly a powerful technique even after all these years.

RBM's are one of the simplest neural network architectures that exist. It consists of just two layers: a visible layer and a hidden layer. It is trained by feeding in training data into the visible layer during a forward pass, and training weights and biases between them during backpropagation. An activation function such as ReLU is used to produce the output from each hidden neuron.

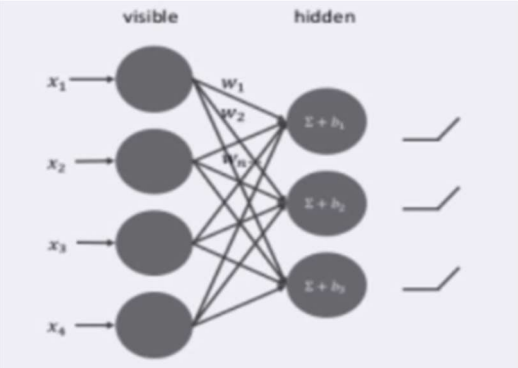

Figure 2: Classical RBM architecture [10]

The reason RBM's have restricted in their name is because neurons in the same layer cannot communicate with each other. However, most modern neural networks also have the same limitation, so the name is a bit dated. The Boltzmann in the name refers to the Boltzmann distribution function used for the sampling function to be explained shortly. RBM's get trained by doing a forward pass, as described, and then a backward pass where the inputs get reconstructed. This is done iteratively over many epochs until convergence to a set of weights and biases that minimizes the error. During the backward pass, the goal is to try and reconstruct the original input by feedback back the output of the forward pass back through the hidden layer, and seeing what values end up in the visible layer. As the weights are initially random, there is typically a big difference between the starting inputs and the reconstructed input. During this process, another set of bias terms on the visible layer get found. To summarize, weights are shared between both the forward and backward passes but there are two sets of biases: the hidden biases used in the forward pass and the visible bias used in the backward pass. Finally, the error between the input and reconstructed input can be measured and the weights can be adjusted using gradient descent to minimize the error.

As you can see, RBM's are conceptually simple. However, there is a challenge that needs to be overcome that once again relates to the sparsity of the data we are working with. Things get a little interesting when RBM's are applied to sparse matrices. How do we train a neural network when most input nodes have no data to work with due to the sparsity? Adapting RBM's for recommending movies on a scale of 1-5 requires a few creative twists. The general idea is to use each individual user in our training data as a set of inputs into the RBM to help train. In other words, each user is processed as part of a batch during training, looking at their ratings for every movie they rated. The visible nodes in the visible layer represent ratings for a given user on every movie and our goal is to learn weights and biases to allow for the reconstruction of ratings for user and movie pairs we have no rating for yet. An important point is that the visible nodes in the visible layer do not take in single input/number. Ratings are categorical data, ranging from 1-5, so each individual rating can be treated as five nodes, one for each possible rating value. For example, if the first rating in our training data is a 5-star rating, the representation for that rating would be four nodes with a value of 0 and one with a value of 1 as shown below. We also have a couple of ratings for

user/movie pairs that are missing and need to be predict. Finally, there is a 3-star rating, represented by a 1 in the third slot and 0's elsewhere. These sample inputs are shown below:

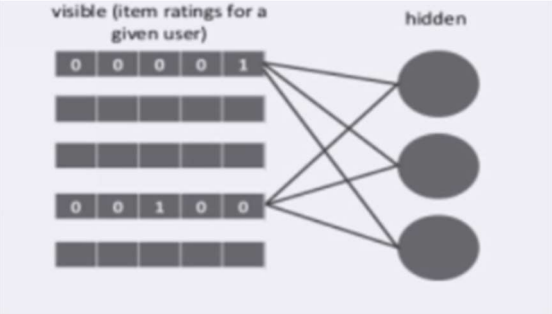

Figure 3: RBM for recommendations [10]

When the RBM is finished training, there will be a set of weights and biases that will allow for the reconstruction of ratings for any user. To predict the ratings of a user for a new movie, we run the RBM using the known ratings of the user we are interested in as the input. The forward pass is run and then the backward pass to end up with the reconstructed rating values for that user. The softmax can then be run on each group of 5 star rating values to transform the output into a 5-star rating for every item. The sparsity presents a challenge because training an RBM on every possible combination of users and movies will have data that is mostly missing because most movies have not been rated by a specific user. The end goal is to predict user ratings for every possible movie so we need to include space for them. This means if a system has N movies, there will be N * 5 visible nodes, and for any given user, most of them are undefined and empty. The solution to this is to exclude any missing ratings from being processed when training the RBM. This is a bit tricky to do as most frameworks, including Tensorflow, assume everything is processed in parallel. However, during training, only the weights and biases used for the movies that the user rated are attempted to be learned. Iterating through training on all users will fill in the other weights and biases. Training an RBM with large amounts of sparse data also requires a bit of ingenuity. The original paper suggests using contrastive divergence, sampling probability distributions during training using a Gibbs sampler. It is only trained on ratings that actually exists, but the resulting weights and biases learned are re-used across other users to fill in the missing ratings for predictions. I wrote an RBM based on the original paper using Tensorflow and can be seen in the file *rbm.py*.

Autoencoders

Using autoencoders, a type of deep neural network architecture, for recommendations was first suggested in the 2015 paper, "AutoRec: Autoencoders Meet Collaborative Filtering" from the Australian National University. The basic architecture of an autoencoder consists of three layers: an input layer on the bottom that contains individual ratings, a hidden layer, and an output layer that gives predictions. A matrix of weights between the layers is maintained across every instance of the network, including a bias node for both the hidden and output layers. The paper trained the network once per movie, feeding in ratings from each user for that movie into the input layer. A sigmoid activation function was used in the output layer. The paper reported slightly better results than RBM's. The architecture is shown below:

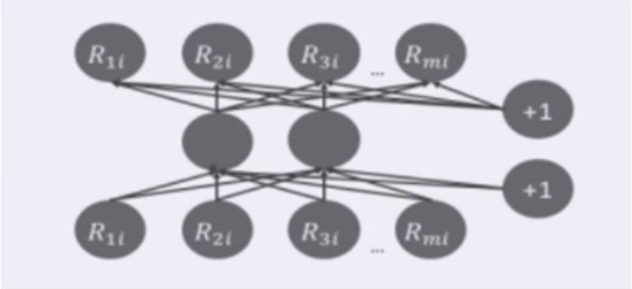

Figure 4: Autoencoder Architecture [11]

Autoencoders also tend to be easier to implement in modern frameworks like Tensorflow, but also must deal with the challenge of sparsity like all collaborative filtering approaches. The paper mentions that they only considered the contribution of observed ratings meaning they were careful to process each path through the network individually, and only propagate information from ratings that exist in the training data, ignoring contributions from input nodes that correspond to missing data from user-movie pairs that are missing. My implementation treats missing ratings as zeroes.

The reason the architecture is referred to as an autoencoder is because the act of building up the

weights and biases between the input and hidden layer which encodes the input by encoding patterns in the input as a set of weights into the hidden layer. The output is reconstructed in the weights between the hidden and output layers which is a decoding process. In summary, the first set of weights is the encoding state and the second set of weights is the decoding stage. My implementation of an autoencoder was written in Tensorflow and can be seen in *autoencoder.py*.

Hybrid Approach

As we saw with the Netflix challenge, there is no need to depend on a single algorithm for a recommendation system. Each algorithm has its own strength and weaknesses, and a better system can result by combining many algorithms together in a hybrid approach. The winner of the Netflix Prize, BellKor, used an ensemble of 107 different algorithms that worked together. The top 2 algorithms used were SVD++ and RBM which ended up being used by Netflix itself. Hybrid approaches can also mitigate the effects of the cold-start problem as combining collaborative filtering with content-based filtering can produce a recommender system that uses user behavior data when it is available and falls back on content attributes when it needs to. The framework I developed makes it surprisingly simple to develop a hybrid algorithm by just combining various algorithm written under the framework. The *hybrid_algorithm.py* file contains the Hybrid_Algorithm class which takes in instances of various algorithms, fits them to the training data, and combines the result to return the final recommendations. Any algorithms can be combined, and based on my experiments, combining an RBM with content-based filtering yielded the best results.

Evaluations

Now that the methods I implemented for making recommendations have been outlined, there needs to be evaluation metrics that can quantify the effectiveness of the different approaches. Evaluations with recommendations are always tricky as users do not care about the rating you think they will give a movie they have not seen. All they care about is having good recommendations. The reason why recommendation systems are as much an art as a science is there is an aesthetic and subjective quality to the recommendations returned. It is hard to consider whether a real person will consider those recommendations good or not, especially when using offline, historical data as it is erroneous to project your tastes onto someone else. This means online, live A/B testing is the best way to test algorithms before they are put into production, but offline evaluation techniques can still be valuable and provide some insight. There are many different ways to measure the quality of a recommender system, and the most popular ones were the ones I implemented and used in the framework. All the metrics discussed below were implemented in *metrics.py* and are automatically calculated when the algorithms are run within the framework.

As in any machine learning task, the concept of training and testing splits is valid in recommendation systems as well. The data would be past user behavior in terms of ratings and the evaluation of the recommendation system would be how well it predicts how people rated things in the past. A better approach than using a single training/test set is to use k-fold cross validation. In k-fold cross validation, instead of using a single training set, many randomly assigned training sets, also known as folds, are used to train the recommendation system independently and the accuracy is measured by using the test set(the rest of the data). Thus, a score of how accurately each training fold ended up predicting user ratings is gathered and an average can be returned. The good thing about this approach is that it prevents overfitting as using a single training set can be dangerous and using cross-validation insures a recommendation system works for any set of ratings, not just the ones that happened to be in the training set chosen.

Commonly used accuracy metrics in literature are Mean Absolute Error(MAE) and Root Mean Squared Error(RMSE), shown next:

$$MAE(r, \hat{r}) = \frac{\sum_{a \in \mathcal{N}_{ui}} |r_a - \hat{r}_a|}{\|r\|}$$

$$RMSE(r, \hat{r}) = \sqrt{\frac{\sum_{a \in \mathcal{N}_{ui}} (r_a - \hat{r}_a)^2}{\|r\|}}$$

The MAE looks at the mean absolute values of each error in rating predictions while the RMSE looks at the square root of the mean squared errors. RMSE tends to inflate the penalty for larger errors which may be desirable in recommendations systems. The RMSE metric was also what had to be improved on in the Netflix challenge of 2006.

The above accuracy metrics are good, but the reality is that accuracy in predicted ratings is not what recommendation systems are meant to do. Real people do not care what the system thinks they should have rated a movie they have already seen and rated. Real people care mostly about what the system thinks the best movies for them to see are which is a different problem from accuracy. The main reason accuracy was an early focus of recommendation systems was the 2006 Netflix Prize where Netflix offered a 1 million dollar prize to the first group to achieve a 10% improvement on Netflix's own RMSE score on a their dataset of ratings. This obviously spurred a lot of interest and research into recommendation systems, and improving accuracy was the main focus. It should be noted that there are not many ways to measure a recommendation system accurately given only historical and offline data. This once again comes to the point that live A/B testing is key and low RMSE scores mean nothing if the recommendations generated by the algorithm for new movies are not well received. In the end, all that matters is the top-N recommendations shown to the user and their reactions to it. There are a few metrics that focus on measuring the quality of top-N recommendations which will be discussed next.

The first metric called, hit rate, is based on generating top-N recommendations for all of the users in the testing set. If one of the recommendations in a users' top-N recommendations is something they rated, that counts as a hit. The idea is that the user was shown something that they had found engaging enough to watch on their own so it should be seen as a success. Adding up all the hits for every user in the test set and dividing by the number of total users results in the hit rate metric.

$$Hit\ Rate = \frac{\#Hits}{\#Users}$$

The way to quantify the hit rate is a method called leave-one-out cross validation. Leave one out cross validation works by first computing the top-N recommendations for all users in the training set, and intentionally removing one of those items from that user's training data. The recommendation system's ability to recommend that item that was left out in the top-N recommendations during the testing phase will count as a hit. It should be noted it is incredibly hard to get one specific movie right for one of the top-N recommendations meaning the hit rate calculated using this method tends to be very low. However, it is still more of a user-focused metric when recommendation systems are designed to produce top-N recommendations in the end, as is the case in the real world. A variation on traditional hit rate is the average reciprocal hit rate (ARHR) which accounts for where in the top-N recommendations the hits appear. More credit is given for successfully recommending an item near the top rather than at the bottom. Since users mostly focus on the beginning of the list, this metric tries to account that tendency.

$$ARHR = \frac{\sum_{i=1}^{n} \frac{1}{rank_i}}{\#Users}$$

Another variation is the cumulative hit rate(CHR). All it does is discard recommendations whose predicted rating is below a specified threshold. The idea is there should not be recommendations for movies if the user may not enjoy that movie. Another twist is rating hit rate(rHR) which breaks down the hit rate by predicted rating score. This gives an idea of the distribution of how the good an algorithm thinks recommended movies are that get hits. As mentioned before, all of these metrics were implemented and built into the framework I developed and can be seen in *metrics.py.*

Focusing on accuracy is not the only metric that matters with recommendation systems. For example, metrics like coverage, diversity, and novelty are important as well. Coverage can be defined as the percentage of users who were given a top-N recommendation whose predicted rating falls above a threshold. For example, if we set a threshold of 4.0, we want to see how many of our users were given a recommendation whose predicted rating is above the threshold. It is measuring how many "good" recommendations we are giving at a high level. Another metric, diversity measures how broad the movies being recommended are. Using the item-item similarity matrix that is generated during item-based collaborative filtering, it is easy to see how diverse the recommendations generated are as it is just the inverse of the average similarity between the movies in the top-N list. It should be kept in mind that high diversity is not always a good thing. High diversity can be achieved by just recommending random items. A final metric is novelty which measures the average popularity ranking of the top-N recommendations. I measured the popularity rankings for the dataset by counting the number of ratings for each movie so movies with higher numbers of ratings can be viewed as being more popular. A high novelty score means the movies recommended are in the long tail of the movie catalog. The long-tail phenomenon is well documented for recommendation systems. The idea is that there are a few popular items that are purchased or watched many times while the vast majority of items fall in the long tail and are not as popular. Thus, a high novelty means the recommendations may be from the long tail which could lead to serendipitous discoveries that is sometimes needed.

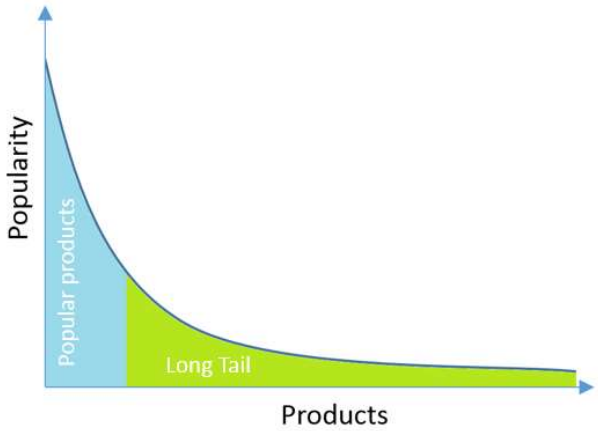

Figure 5: Long tail phenomenon

Which metric matters more than anything? The answer is none of them as the true measurement of how good a recommendation system can only be captured by live, online A/B tests on real users. That is the only way to measure how real people react to recommendations put in front of them. It is impossible to simulate what goes on inside the heads of real people when they see new things meaning it is imperative to try out algorithms on them in real world, in a controlled manner, to see what works and what does not. However, the metrics mentioned above can give a sense of how good an algorithm might perform relative to others and they are the best metrics for offline data.

The evaluations are summarized in the table and the training and validation loss for the RBM is shown in Figure 6:

	MAE	RMSE	HR	ARHR	Coverage	Novelty	Diversity
Random	1.1478	1.4385	0.0089	0.0014	1.0000	557.8365	0.7910
Content based(Year and Genre)	0.7421	0.9542	0.0055	00.0032	0.8823	3937.8348	0.5192
Content-Based(Mise en Scene)	0.8364	1.0633	0.0015	0.0004	0.7720	4197.5798	0.6929
Item-based	0.7798	0.9995	0.0502	0.0213	0.9896	6740.0338	0.6495
User-Based	0.7798	0.995	0.0554	0.0347	1.0000	5654.1042	0.8586
SVD	0.6958	0.9002	0.238	0.0108	0.9300	472.7290	0.0431
SVD++	0.8928	0.6865	0.0328	0.0143	0.9434	666.1976	0.0947
RBM	0.9935	1.1897	0.0030	0.0022	0.9532	917.8996	0.0180
Autoencoder	1.4222	1.8253	0.0075	0.0026	1.0000	512.4595	0.0713
Hybrid	0.9929	1.1890	0.0039	0.0019	0.9943	1992.4049	0.2059

There are a few interesting results that turn up in my evaluations. First, content-based filtering with year and genres has lower RMSE and MAE scores than all the other approaches but the matrix factorization

ones. However, they had low hit rates. This once again shows how focusing on accuracy metrics like RMSE and MAE may not be the best idea. Also, item based collaborative filtering performed slightly worse than user-based collaborative filtering. This contradicts industry trends where Amazon has used item-based to great effect. It is likely this could be attributed to the dataset being used as it is not particularly large and using historical, offline data to evaluate recommendation systems is always a tricky task. In addition, the evaluations for RBM and Autoencoder were worse than all the others in pretty much every category. However, I cannot state enough that these algorithms need to be tested on real people in real systems and the dataset may not be large enough to yield the best results for deep neural networks. In addition, the way both algorithms work tends to lead to conservative ratings meaning the predicted ratings tend to be low. This could lead to worse RMSE and MAE scores that are deceptive. The matrix factorization methods performed admirably in RMSE and MAE scores and had good hit rates as well. This is not surprising considering the effectiveness of matrix factorization in many real-world recommendation systems. In addition, the RMSE and MAE scores for the hybrid model were between the content-based and RBM ones which formed the components of the hybrid model I created.

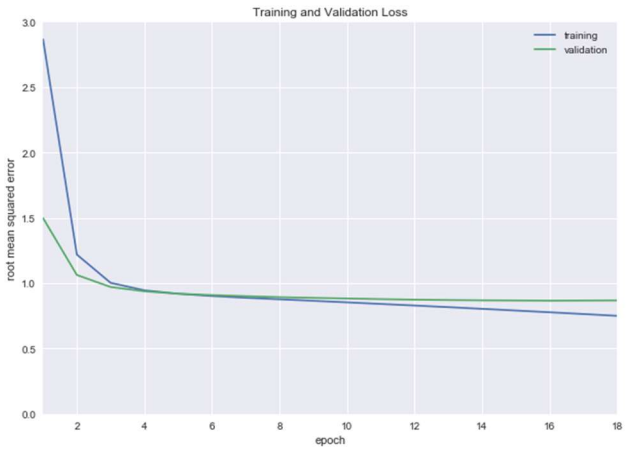

Figure 6: RBM training and validation loss

Scalability

The ratings data I worked with for this project is relatively small. In real-world systems, there would be hundreds of millions of ratings with millions of users and hundreds of thousands of movies. The algorithms I wrote and tested would work just fine and probably better with more data, but many of these algorithms take a long time to train and run which could be problematic in a real-world setting. A solution is to use Apache Spark. Apache Spark is a very powerful for processing massive datasets across a cluster of companies. It makes it very easy to process massive datasets with just a few steps. The first step is to write a driver script that defines how the data will be processed using Spark API's. Once the driver script is launched, it will communicate with the cluster manager to allocate the resources it needs from the clusters. The cluster manager will allocate executor processes across the cluster that will do the actual work of processing the data in the most efficient manner. For us software developers, Spark is essentially just a core that manages the distributed computing and libraries that facilitate the programming such as MLLib, Spark's machine learning library. MLLib contains classes that make recommendation generation from massive data sets very simple. My *SparkALS-20m.py* file contains the script that runs Alternating Least Squares for matrix factorization on the dataset and evaluates the RMSE. Apache Spark already includes many built-in recommendation algorithms, but a future work could be to make extend my framework to use clusters when running.

There are a few other open-source tools worth mentioning that aid generating recommendations for massive datasets. Amazon has a system called Deep Scalable Sparse Tensor Neural Engine(DSSTNE). In 2016, Amazon open-sourced the system and it is currently used by Amazon itself to generate recommendations. DSSTNE makes it possible to set up deep neural networks using sparse data without writing any code. All that is required is to write a configuration file that defines the topology of the neural network and how it will be optimized. DSSNTE also runs on a GPU meaning massively parallel data processing can be done very quickly.

Another tool is Amazon Sagemaker, a component of Amazon Web Services, which allows for the creation and training of large-scale models in the cloud and

for predictions to be made from that model quickly in the cloud as well. This is essentially what Amazon and Netflix do as they want to generate recommendations in an on-demand, responsive manner.

Challenges

During my research, I came across and read about many challenges that are unique to recommendation systems that are worth a mention.

One of the most well-known issues with recommendation systems that I have already mentioned a few times is the cold start problem. If a brand-new user joins a service, what recommendations would be appropriate for them if there is no information about them. This also applies to new items that are added to a catalog. If no one has watched or purchased that item, how can any meaningful recommendation of that item be made to someone? Typically, a new user will eventually watch a movie or make a purchase, so information will be available at some point. This situation is where implicit data can be a gold mine. If a user clicks on a movie or views/purchases an item, that click or view can be interpreted as implicit interest and can be used for recommendations. In addition, using implicit data along with content-based filtering has seen great success as seen in Amazon's "people who bought this also bought" feature. Another solution is to just recommend the most popular items to a new user. Popular items are popular because they appeal to a large segment of the population so recommending one is a safe option. Some systems like Goodreads ask users to rate 20 books they have already read along with marking their favorite genres once they join the service. I believe the cold-start problem is not as big of an issue anymore due to the plethora of implicit data that can be used.

Another challenge with recommendation systems is that sometimes, you may not want to recommend certain items. For example, some topics may be offensive and are to be avoided. Thus, care has to be taken by using stop-lists to filter out these offensive recommendations before they make it to the user. Stop-lists are a very simple concept, containing words or topics that are offensive so movies matching those criterium should not be recommended at all.

Filter bubbles are another challenge with recommendation systems. Filter bubbles refer to societal problems that arise when all the recommendations to a user appeal to their existing interests. For example, if someone who watches right-wing politics movies, they are likely to be recommended those types of movies in the future. On the other hand, if someone watches a lot of left-wing movies, they will be recommended those types of movies in the future. Thus, their beliefs will be reinforced and strengthened by recommendations and they may not see different viewpoints or areas. This is not bad from a business standpoint, but is bad from a ethical standpoint. A little extra diversity in recommendations can help expose people to different ideas to break them out of their bubble. This is where recommended a couple random items, known as random exploration, can help as it prevents recommendations from being stuck in a bubble.

Trust is also an important challenge in recommendation systems. It is good to make sure that in the top-N recommendations, a few names are recognizable to ensure confidence in the system. If all the recommendations are obscure, people might not trust the recommendation system even if those obscure recommendations are movies they might like. Neighborhood-based collaborative filtering works well in establishing trust as the results returned are intuitive and transparent. A system can say a movie was recommended because users similar to you liked it or the movie is similar to movies you have liked.

A final challenge that is not unique to recommendation systems and occurs across all data science and machine learning problems is dealing with outliers and data cleaning. The results of a data science project are only as good as the data fed into it so data preprocessing is a key step in all machine learnings pipelines. Cleaning data for recommendation systems could involve making sure ratings are from real people and not bots. In addition, people who review items for a living will generate many reviews, but their opinion should not necessarily be more powerful than everyone else. A person with many reviews will have an outsized

influence on the recommendations a system makes so it is important to consider this when designing a system.

Future Work

Much of the recent work on recommendation learning has focused on using deep learning and other advanced methods, and future work in this area will likely continue this trend. I have highlighted a few recent papers and techniques below that show promise and could form integral part of future recommendation systems.

A paper from the 2017 Conference on Recommendation Systems from a team at the University of California, San Diego introduced the concept of translation-based recommendations [14]. The idea is that users are modeled as vectors moving from one item to another in a multi-dimensional space, and the sequence of events – like which movie a user is likely to watch next – can be predicted by modeling these vectors. This method shows promise as it out-performed all of the best existing methods for recommending sequences of events in all but one case in one data set. They also focused on hit rate which is a more relevant metric for real world recommendation systems. The way it works is that individual movies are positioned in a "transition space" where neighborhoods within this space represent similarity between items, so items close together in this space are similar to each other. The dimensions correspond to complex transition relationships between items. For example, a user who watches a Nicolas Cage movie is likely to watch a Nicolas Cage movie next. This transition would be represented by a vector in this space meaning the next movie the user is likely to watch can be extrapolated along the vector associated with the user.

In 2011, the University of Minnesota published a paper called "SLIM: Sparse Linear Methods for Top-N Recommendation Systems" [16]. The University of Minnesota has been a pioneer in the field of recommendation systems from the very beginning and this paper continues that trend. SLIM's results on a wide variety of data sets against competing algorithms was impressive. In addition, success was measured by hit rate which has more meaning for top-N recommendations. SLIM beat all algorithms on the Netflix dataset, a data set from a book store, Yahoo! Music data, and a couple mail-order retailer data sets. On the MovieLens dataset, it finished second. The idea behind SLIM is to generate recommendation scores for a given user and item by a sparse aggregation of the other things that the user has rated, multiplied by a sparse aggregation of weights.

Another interesting direction for future work is to look at session-based recommendations where recommendations can be made even if a user is not registered or known to the system. It is well-documented that recurrent neural networks are good at finding patterns in sequences of data and predicting what will come next. For example, assume you are viewing videos on Youtube and you are not logged in, so the website has no past information on you, but it will know the sequence of video you have watched and RNN's can be used to predict what video you will watch next. The paper that describes using RNN's for making session-based recommendations is called "Session-Based Recommendations with Recurrent Neural Networks", published in 2016 []. The paper is mostly concerned with how to tweak RNN's to work well with session-based clickstream data and relies heavily on gated recurrent units(GRUs). Their system is called GRU4Rec and they released a Theano implementation. For educational purposes, I ported the software to Tensorflow and tweaked it to treat the MovieLens data as a clickstream, using the timestamps of ratings as a approximation of a clickstream. The results were not the best, but not bad considering the MovieLens data is not session-based in the first place.

A final interesting future direction is using deep factorization machines. The 2017 paper "DeepFM: A Factorization Machine based Neural Network for CTR Prediction" from a team at the Harbin Institute of Technology in China, looks at this approach [12]. The main idea behind this approach is to combine the strength of factorization machines and deep neural networks. Factorization machines are a generalization of matrix factorization. They essentially do the same thing as SVD, but they can take in categorical data and can find latent

relationships between any combination of features they are given. This makes them more general purpose than SVD, finding relationships SVD would not have considered. Also factorization machines are best at finding lower-order feature interactions while deep neural networks excel at finding higher-order feature interactions. DeepFM promises the best of both worlds. The idea is that the training data is fed in parallel to both a factorization machine and a deep neural network. The outputs from the factorization machine and the deep neural network are combined into a final sigmoid unit that predicts whether the user is likely to watch, purchase, or click on an item. DeepFM is nothing more than a hybrid approach and the fact that it is inspired by two very successful approaches, matrix factorization and deep learning, its potential is appealing.

Another area of future work I came across while researching was how to incorporate temporality into recommendation systems. For example, when watching Netflix during Christmas time, Netflix may want to recommend holiday movies at a higher rate. Also, horror movies might be more popular during the night hours, so they could be recommended at higher rates during those times. Incorporating seasonality and temporality could lead to more relevant recommendations based on context.

Conclusion

Recommendations systems are currently a hot topic and will continue to grow in importance. Traditional approaches like neighborhood-based collaborative filtering have done well in the past, but model-based approaches using matrix factorization and deep learning are becoming the norm in recommendation systems. Hybrid approaches promise the best result as leveraging the strengths of various algorithms is an intuitive approach that has worked well in many areas of machine learning. I once again want to iterate that evaluating recommendation systems offline is not an easy task so the results I gathered may not necessarily be the best or most intuitive. However, it cannot be repeated enough that offline testing is not the norm in recommendation systems. Discarding an algorithm because of a higher RMSE score or even a lower hit rate would be a grave error. The only way to be sure which algorithms work the best in a real-world setting is to test them on real world users through A/B testing in a controlled manner. In the end, the future of research into recommendation systems looks promising and we can all look forward to the day when everything we do is a recommendation.

References

[1] Desrosiers, Christian, Karypis, George. 2011. “A comprehensive survey of neighborhood-based recommendation methods”. University of Minnesota

[2] Harper, Maxwell F. and Konstan, Joseph. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4, Article 19 (December 2015), 19 pages. DOI=http://dx.doi.org/10.1145/2827872

[3] Herlocker, Jonathan L. et al. “Evaluating collaborative filtering recommender systems.” ACM Trans. Inf. Syst. 22 (2004): 5-53.

[4] Levinas, Adrian Claudio. 2014. “An Analysis of Memory Based Collaborative Filtering Recommender Systems with Improvement Proposals”

[5] Linden, Greg et al. “Industry Report: Amazon.com Recommendations: Item-to-Item Collaborative Filtering.” IEEE Distributed Systems Online 4 (2003): n. pag.

[6] Resnick, Paul et al. “GroupLens: An Open Architecture for Collaborative Filtering of Netnews.” CSCW (1994).

[7] Schafer, J. Ben et al. “Collaborative Filtering Recommender Systems.” The Adaptive Web (2007).

[8] Wei, Jian et al. “Collaborative filtering and deep learning based recommendation system for cold start items.” Expert Syst. Appl. 69 (2017): 29-39.

[9] Bell, Robert M., Yehuda Koren and Chris Volinsky. “The BellKor solution to the Netflix Prize.” (2007).

[10] Salakhutdinov, Ruslan et al. “Restricted Boltzmann machines for collaborative filtering.” *ICML* (2007).

[11] Sedhain, Suvash, Krishna Menon, Aditya & Sanner, Scott & Xie, Lexing. (2015). AutoRec: Autoencoders Meet Collaborative Filtering. 111-112. 10.1145/2740908.2742726.

[12] Guo, Huifeng, et al. "DeepFM: a factorization-machine based neural network for ctr prediction." *arXiv preprint arXiv:1703.04247* (2017).

[13] Hidasi, Balázs, et al. "Session-based recommendations with recurrent neural networks." *arXiv preprint arXiv:1511.06939* (2015).

[14] He, Ruining, Wang-Cheng Kang, and Julian McAuley. "Translation-based recommendation." *Proceedings of the Eleventh ACM Conference on Recommender Systems*. ACM, 2017.

[15] Covington, Paul, Jay Adams, and Emre Sargin. "Deep neural networks for Youtube recommendations." *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM, 2016.

[16] Ning, Xia, and George Karypis. "Slim: Sparse linear methods for top-n recommender systems." *2011 11th IEEE International Conference on Data Mining*. IEEE, 2011.