MovieLens Recommender System Project Report

CONTENTS

INTRODUCTION

Automated recommendations are everywhere: Netflix, Spotify, Google, and more. Recommender systems learn about a user's unique interests and show the products or content the system thinks (s)he will like best. In this project, I build off of Frank Kane's "Building Recommender Systems with Machine Learning and AI" course where he covers recommendation algorithms based on neighborhood-based collaborative filtering, matrix factorization, neural networks, and more. Before embarking on this project, I read numerous articles, watched hours of videos, and completed two other recommender system tutorials. I ultimately chose to build off of Frank's framework because of the depth and breadth of his course. Furthermore, because of his extensive industry experience, he taught it from a business perspective and explained the limitations of certain metrics and algorithms. It was this unique and holistic view that convinced me to make recommenders from Frank's framework.

This is how my project is broken down: I introduce the MovieLens dataset and perform explanatory data analysis on it. After, I'll go over Frank's Python framework for generating and evaluating recommendations for the different models. It is this framework that will allow me to implement and evaluate new ideas. Next, I build recommenders using the different types of algorithms. Content based models

recommend items to people just based on the attributes of the items themselves. Neighborhood based collaborative filtering is a behavior-based recommendation approach. Then, I build recommenders using more modern methods that generally rely on matrix factorization, followed by deep learning. I experiment with Hybrid approaches before picking my best model and concluding with recommendations and areas for further exploration.

EXPLANATORY DATA ANALYSIS

How do recommender systems work? It comes down to understanding people. A recommender system starts with some information about a person that it uses to figure out his/her individual tastes and interests. Then, it merges its data about him/her, with the collective behavior of everyone else like him/her, to recommend stuff a person might like. But where does that data about a person's unique interests come from? One way to understand users or customers is through explicit feedback, which in this project, are movie ratings on a scale of one to five stars. In this case, a company explicitly asks users to do this, and the garnered data is used to build up a profile of that user's interests.

This is the case for the MovieLens dataset I use for my recommender systems. It contains 100,000 ratings for 9,000 movies by 600 users. Frank used the 2016 version during his course, but I downloaded the most recent 2018 version. It's essentially the same dataset but it's updated to include movies up to 2018. I also wanted to use the most recent dataset as a personal challenge because results Frank got in his 2016 will not be the same for my dataset.

There are four different csv files in the MovieLens dataset, and although I only use two of them, I will briefly explore all four here. The first is the "Links.csv" file, and it contains 9,742 rows and 3 features. Each of the features correspond to the MovieLens identifier (MovieId), The Movie Database identifier (tmdbId), and the Internet Movie Database (imdbId). Frank does not use this dataset, but it is useful for future exploration and expansion as these "keys" are what link MovieLens to the other movie datasets. Those datasets have

other information such as summaries, actors, and directors. The sky is the limit when we have such a vast library of information to play with!

"Tags.csv" is also not used. It contains 3,683 rows and four columns of information. It has userId for each individual and a movieId for the "tags" each user gave. These "tags" are either a word or short phrase used to describe a movie. The last column represents the timestamp of each tag. All in all, 58 users gave tags to 1,572 movies.

The "movies.csv" contains 9,742 rows and three features which represent the unique movie identifier, the title, and their respective genres. It had only 5 duplicates and I would have removed them, but I did not want to possibly break Frank's code. Furthermore, because it represented the tiniest fraction of the dataset, I just left it alone.

The "ratings.csv" contains 100,836 rows and four columns representing the user ID, movie ID, rating, and the timestamp of the rating. It has 610 users who gave a majority rating of either 4 stars or 3 stars. This amounted to almost 50% of the ratings.

In section 3.3, I merge the "movies.csv" and "ratings.csv" dataset to place all my data in one DataFrame. With almost 7600 ratings, most movies had less than 3 ratings each. The movies with the most ratings had an average release date in the 1990s, and Forrest Gump had the most ratings with almost 330. Most movies either belonged to the Drama or Comedy genre, and 8 users had over 200 ratings with the top user having over 700!

Section 3.6 is where I found a user with similar movie preferences to my own. Frank's framework prints out top 10 recommendations, and because "User 25" has rated movies I loved, it adds a qualitative dimension to assessing the performance of the recommender. "User 25" highly rated movies such as the "Lord of the Rings" trilogy, Avengers, Star Wars, and similar content. I look forward to seeing what movies my algorithms will recommend for this user.

TOP-N RECOMMENDER AND EVALUATION METRICS

The type of recommendation system I'll be making is called a called a "top-N recommender system". This means that the recommender will produce a finite list of the best things to present to a given person. When starting my own research into this topic, a lot of what I saw tended to focus on the problem of predicting a user's ratings for everything they haven't rated already. Unfortunately, that's very different from what recommender systems do in the real world as users don't care about an algorithm's ability to predict how they'll rate some new item. They just want to see things they're likely to love. My ultimate goal, for this project, is to put the best movies in front of users in the form of a top-10 list. My success depends on the ability to find the best top recommendations for people so I will focus on finding things people will love and not the ability to predict the items people will hate. In the end, it's those top five or 10 recommendations that really matter.

Frank says, "That a big part of why recommender systems are as much an art as they are science is that it's difficult to measure how good they are." There's a certain aesthetic quality to the movie recommendations and it's hard to decipher if a user will like them, especially if the algorithm is not being A/B tested online. There are a lot of different ways to measure the quality of a recommender system and often different measurements can be at odds with each other but the following section explains the more popular metrics for recommender systems, as they all have their own uses.

First, the Mean Absolute Error or MAE is the mean or average absolute values of each error in rating predictions, and I want to have the lowest number (error). A slightly fancier metric is the Root Mean Square Error, or RMSE. This is a more popular metric for a few reasons, but one is that it penalizes more when a rating prediction is way off, and penalizes less when the rating prediction was reasonably close. The difference is that instead of summing up absolute values of each rating prediction error, the RMSE sums the squares of the rating prediction errors instead. This inflates the penalty for larger errors, and then it takes the square root of the average of the errors.

Another metric is called Hit Rate, and measuring it is as follows: First, I generate top-10 recommendations for all of the users in the test-set. Then, if one of the recommendations in a user's top 10 recommendations is something they actually rated, it is considered a hit. Last, add up all of the hits in the top-10 recommendations, for every user in the test-set, and divide by the total number of users. That is how to calculate hit rate.

Hit rate itself is easy to understand but tricky to measure, so a clever way around this is using leave-one-out cross validation. How this works is that an algorithm computes the top-10 recommendations for each user in the training data, and intentionally removes one of those items from that user's training data. Then, for each user, test the model's ability to recommend that item that was left out of the top 10 results. Unfortunately, it's a lot harder to get one specific movie right than to just get any one of the ten recommendations. This means hit rate with leave-one-out cross validation tends to be very small and difficult to measure, unless one has a very large dataset to begin with. The reason why it's a metric is because it's more user-focused metric.

A variation on hit rate is Average Reciprocal Hit Rate, or ARHR for short. This metric is just like hit rate, but it accounts for where in the top-10 list the hits appear. I get more credit for successfully recommending an item in the top slot than in the bottom slot. This is useful because, in the real world, users tend to focus on the beginning of the list of recommendations. Instead of summing the number of hits, ARHR sums up the reciprocal rank of each hit.

Another twist is Cumulative Hit Rank (CHR), and it throws away hits if the predicted ratings lie below some threshold. The idea is that a model shouldn't get credit for recommending items to a user that they might not actually enjoy. For example, if there was a cutoff of three stars, the CHR wouldn't count any top-10 recommendations with less than three stars.

Accuracy isn't the only thing that matters with recommender models as there are other ways to evaluate the system. One is Coverage and that's the percentage of possible recommendations that recommender is able to provide. It's worth noting that coverage can be at odds with accuracy because the

model enforces a higher quality threshold on the recommendations, then it might improve accuracy at the expense of coverage. Finding the balance can be delicate.

Another metric is called Diversity. This is as a measure of how broad a variety of items the algorithm is recommending. Many recommender systems start by computing some sort of similarity metric between items. After computing the similarity scores of every possible pair in a list of top 10 recommendations, the algorithm averages them to get a measure of how similar the recommended items in the list are to each other. Diversity is basically the opposite of Average Similarity, so it is subtracted from one to compute diversity.

Diversity, in a recommender system, isn't always a good thing as someone can achieve high diversity just by recommending completely random items. Unusually high diversity scores could be an indication of bad recommendations, and it’s a good idea to weigh diversity alongside metrics that measure the quality of the recommendations as well.

Similarly, Novelty is a measure of how popular the items are that is being recommended, and like diversity, just recommending random stuff would yield very high novelty. Although novelty is measurable, what to do with it is subjective. There's a concept of user trust in a recommender system, and people want to see at least a few familiar items in their recommendations. If an algorithm only recommends things people have never heard of, they may engage less with the recommendations as a result. By contrast, popular items are usually enjoyable by a large segment of the population, so they’re a safe bet for users who haven’t watched them yet. If the algorithm is not recommending some popular items, one should question the validity of the system.

The algorithm needs to strike a balance between familiar popular items and new items the user has never heard of before. The familiar items establish trust with the user and the new ones allow the user to discover entirely new things that they might love. Novelty is important though because the whole point of recommender systems is to service items in what Frank calls, “the long tail”. Most sales come from a very small number of items, but the rest of the items, in the long tail, makes up a large number of sales as well. If modeled successfully, these niche items have the potential to increase revenue for the company.

Building recommender systems is a bit of an art and the all the metrics used to evaluate the system needs to be taken contextually and holistically. In the end, there are a lot of different ways to evaluate a recommender system, and what should one focus on? It depends on many factors but it usually boils down to what the business is trying to achieve, and since it's most probably trying to make money, this is arguably the most important factor of all.

For this project, I will focus on recommending movies a specific user would love. The business context is that if users love the movies being recommended to them, then it will keep them subscribed to the platform.

FRAMEWORK

In this section, I will introduce Frank's core framework for building recommender systems. He wrote several custom python modules to automate data ingestion, processing, and training and testing of the models. They five are the foundation for all the recommenders I build.

RecommenderMetrics.py is the first python script and it is where all the metrics, explained in the last section, are created. Some of the metrics like RMSE and MAE have been imported from Surpiselib which is a Python scikit for building and analyzing recommender systems. Others, such as all the Hit Rate metrics were coded by Frank himself. The GetTopN function on line 14 is how the top 10 recommendations are generated.

The second framework script is MovieLens.py. It's responsible for loading up the raw MovieLens files that contain ratings and information about the movies, and converting them into datasets that Surpiselib can use. It also includes some useful functions for extrapolating information such as movie title, genre, and movie release year. Because I adapted the code to run on Google Colab, I had to comment out line 22 as it kept changing the home directory.

The third module is EvaluatedAlgorithm.py, and it is essentially a wrapper for a bunch of calls to the RecommenderMetrics.py functions. The Evaluate method, on line 16, has the option to calculate the Top-N,

hit rate, diversity, and novelty metrics, so one can avoid the expense and time of computing these if all (s)he is only interested in the RMSE and MAE.

The fourth module is EvaluationData.py and it generates all the various train_test_splits required by the RecommenderMetrics.py functions. It essentially pre-processes the data for modeling.

The fifth module is Evaluator.py and it is the highest-level interface that hides the complexity of the other modules. Line 18 has the AddAlgorithm method which makes it easy to add models for training. The SampleTopNRecs method, on line 55, retrieves the top 10 recommendations for a specific user which is specified in this function.

When taken together, these five modules serve as the foundational framework which makes building, testing, and evaluating recommender systems easier and fun.

CONTENT BASED RECOMMENDERS

Content-based systems recommends items based on the attributes of those items themselves, instead of trying to use aggregate user behavior data. The MovieLens dataset doesn't give much content to work with, but one thing it does give is which movie genres each movie belongs to. Thus, if a user likes science fiction movies, it's reasonable to recommend other science fiction movies to him. MovieLens also has movie release years, so that information can be used as well. Instead of just recommending all science fiction movies to a user who likes science fiction, the algorithm can be narrowed down further to science fiction movies that were released close to the same year as the movies this person liked.

This gets complicated when using multiple genres, and one approach that works really well in a lot of different situations is the cosine similarity metric. In essence, it measures similarities, from the movies a user rated, to the other movies in the database. The metric then scores the similarity with a value between 0 and 1, 1 being exactly similar.

Aside from genres, the other factor used was release years. When designing a similarity metric, Frank said he got creative. He took the absolute difference between the release years of pairs of movies, and used

an exponential decay function to turn the data into a range of values between 0 and 1. At a year difference of zero, the similarity score is 1, and it decays exponentially, getting pretty small at around a difference of 10 years, and almost nothing at 20.

Next, Frank uses the KNN approach to measure the content-based similarity between everything a given user has rated, and the rating prediction for a movie. In this project, the algorithm will select the 40 movies whose genres and/or release dates most closely match the movie to be evaluated for User 25. In essence, KNN is just selecting some number of things that are closely related to the thing a user is interested in, and predicting something about that item based on the properties of its neighbors.

All of this is coded in the ContentKNNAlgorithm.py file. Line 28 calls the getGenres method from the MovieLens module. It is the same on Line 29 for the getYears function. Line 46 is where I decide what content I want to use for similarity scores. I can use movie genres, release years, or both as factors for similarities.

In section 5.4.3, I noticed a quirk that bugged me. When using only the release years for my algorithm, it tended to recommend really old movies, two of which came from the 1920s and 1930s. However, when I changed a line of code as seen in section 5.4.4, it started recommending very new movies from 2018 (which is the latest year in the dataset). In the end, I chose that as my best content-based recommender as I preferred it to recommend newer movies over older ones.

COLLABORATIVE BASED RECOMMENDERS

Collaborative Based Recommenders leverages the behavior or others to inform what a user might enjoy. At a very high level, it means finding other people like him/her and recommending stuff they liked. Or it might mean finding other things similar to the things that he/she likes. Either way, the idea is taking cues from people similar to a specified user and recommending stuff based on the things they like that this user has not seen yet. It's recommending stuff based on other people's collaborative behavior.

The first step in collaborative recommenders is measuring the similarity between things or the similarity between people to find others like a specified user. The cosine similarity metric works really well in

most cases. The only difference between this how it was used with content recommenders is that the dimensions are based on user behavior, instead of content attributes.

The big challenge in measuring these similarities, based on behavior data, is the sparsity of the datasets. There are so many movies in the world, that it's very unlikely that an individual person has seen, or even rated, every specific movie. This means that it's tough for collaborative filtering to work well, unless there's a lot of user behavior data to work with. This is why collaborative filtering works well for big companies because they have enough data to generate meaningful relations, in spite of the data sparsity. Frank says the 100,000 ratings dataset isn't large enough to generate really good similarity data.

Other metrics for measuring similarity are the Adjusted Cosine and Pearson which are two terms for basically the same thing. It's mean centered cosine similarities and it works in average rating behavior across all of the user's item ratings, or the average ratings of an item across all users. The idea is to deal with unusual rating behavior that deviates from the mean. Surpiselib refers to adjusted cosine as "user-based Pearson similarity." By contrast, Pearson similarity is what Surpiselib calls "item-based Pearson similarity." The names are different but it is essentially the same thing, and it's convenient that Surpiselib offers this similarity metric so I don't have to code it myself.

Mean Squared Difference (MSD) is computing the mean of the squared differences between how each user rated each item. It's directly comparing how two people rated the same set of things, and it's similar in concept to how MAE works when measuring the accuracy of a recommender system.

There are two different ways of doing collaborative filtering: Item-based and user-based. It's important to remember that most of these similarity metrics can apply to either approach. The idea behind user-based collaborative filtering is pretty simple. In this project, I start by finding other users similar to User 25, based on their ratings history, and then recommend stuff they liked that User 25 has not seen yet. Frank points out that saying two users are 100% similar could also mean that they hate the same things. In fact, in a sparse data situation, two users could end up being 100% similar even if they just have 1 movie in common. This issue can

be solved by enforcing a minimum threshold on how many movie users have in common before considering them.

Item-based collaborative filtering is looking at the things User 25 liked, and recommending movies similar to those. We call this item-based collaborative filtering, instead of user-based. There are a few reasons why using similarities between items could be better than similarities between people. One, is that items tend to be of a more permanent nature than people, whereas an individual's tastes may change very quickly over time. Focusing on unchanging items can produce better results when compared to looking at similarities between people, who may have liked something last week, and something totally different this week. Another very important advantage to building item similarities is that there are usually far fewer items to deal with than people. This makes it easier and faster to compute similarity matrices, which is important in the real world. Furthermore, using item-similarities also makes for a better experience for new users. It's easier to compare a similar movie to one (s)he likes, whereas with user-based collaborative filtering, the user would not have any recommendations at all.

Because these similarity metrics and KNN approach are built into Surpiselib, and incorporated the Frank's core framework, it was really easy to experiment with different models. Section 6.3 has 14 different item-based and user-based collaborative recommenders. Although I used different similarity metrics, the results were not all that different from each other. The collective RMSE hovered around 0.90 and all the models had very high Novelty scores. Even worse, none of the actual movie recommendations were recognizable. I would not recommend this recommender approach with this limited dataset.

MATRIX FACTORIZATION METHODS

Frank says collaborative filtering can produce great results when used in large scale situations. However, a legitimate problem with collaborative methods is that it's sensitive to noisy data and sparse data, and one can only get really good results if it's large data set that's nice and clean. Another approach is what Frank calls, "label model-based methods," and instead of trying to find items or users that are similar to each

other, data science and machine learning techniques are applied to extract predictions from the ratings data. The approach is to train models with user-ratings data, and use those models to predict the ratings of new movies by the users. This is exactly the type of architecture that Surpiselib is built around.

It is essentially repurposing machine learning algorithms to build recommender systems, some of which are very good at predicting ratings. There are a wide variety of techniques that fall under the category of matrix factorization, and these algorithms find broader features of users and items on their own. The general idea is to describe users and movies as combinations of different amounts of each feature.

This comes back to the data sparsity issue where a user-item matrix with rows representing users, and columns representing movies. The problem is most of the cells in this matrix are unknown, and the challenge is to fill those unknown cells in with rating predictions. One such technique is matrix factorization and it predicts missing rating values by breaking down the original user-item matrix into smaller matrices that can be recombined with rating predictions.

One very effective and popular algorithm for doing this is called Singular Value Decomposition or SVD for short. If one thinks of all ratings for a set of users and items as a matrix R, it can be factored into smaller matrices that describe general categories of users and items that can be multiplied together. SVD does this quickly. With those factored matrices, the model can predict the rating of any item by any user by just taking a dot product from each matrix. Techniques such as SGD, stochastic gradient descent, and ALS, alternating least squares, can be used to learn the best values of those factored matrices to impute missing ratings.

Surpiselib includes a couple of different SVD implementations. One is SVD and the other the other SVD++, which is a slight variant of SVD that ended up being part of the winning system for the Netflix prize. I used both of these in section 7.3. I tuned SVD using RandomizedSearchCV to find my hyperparameters. Unfortunately, in section 7.3.2, neither RandomizedSearchCV nor GridSearchCV worked when attempting to tune SVD++. I borrowed the SVD hyperparameters and used those for SVD++.

Interestingly, the results were similar for both tuned and untuned versions of SVD++. The RMSE did decrease a little with the tuned models, but it also inflated the Novelty scores. In contrast, the untuned

versions has lower Novelty scores which I prefer. However, the top-10 recommendations were pretty good with all 4 of the different models. I ended up using all of them in the Hybrid section.

DEEP LEARNING

Artificial intelligence, deep learning, and neural networks are the big buzzwords now so it's not surprising that there's a lot of research around applying deep learning to recommender systems. Matrix factorization can be implemented with a neural network, but SVD already does a good job of recommendations. However, deep learning also opens up entirely new approaches to making recommendations that are worth exploring. Frank says, “the main reason to experiment with applying neural networks to recommender systems is that it lets us take advantage of all the rapid advances in the fields of AI and deep learning”. Furthermore, using neural networks with TensorFlow, in a cluster, allows recommenders to utilize a whole fleet of GPUs. Because the technology is proliferating in this direction it’s worth experimenting with it.

The grandfather of neural networks for recommender systems is the Restricted Boltzmann Machine or RBM for short. It's been in use since 2007 and was part of the winning algorithm (along with SVD++) to win the Netflix prize. RBM's are really one of the simplest neural networks containing just a visible layer and a hidden layer. It’s trained by feeding the data into the visible layer, in a forward pass, and optimizing weights and biases during back propagation. An activation function such as ReLU is used to produce the output from each hidden neuron. This is done iteratively over many epochs until the model converges on a set of weights and biases that minimizes the error.

All of this Frank coded in his RBM.py file introduced in section 8.3.1. This is what implements the RBM itself and it uses tensor flow so the model can be trained using the GPU. Section 8.3.2 introduces the RBMAlgorithm.py and it essentially the wrapper for the RBM.py module which makes it both easier to use as well as ties into Frank’s core recommendation framework.

When trying to tune the RBM, I ran into issues. I had to update the code so that it could run with Tensor Flow 2 as the model was originally written for Tensor Flow 1. Also, I tried to optimize a set of Hyperparameters using RandomizedSearchCV but it would not work. With a very basic set of hyperparameters, using GridSearchCV with five-fold cross validations, it took overnight to train. If I had more time and more resources, I would converge on a better set of hyperparameters.

The model itself was underwhelming. The lower RMSE was negligible compared to the untuned version, but the Hit Rates were better. However, there is a marketable difference between the Novelty scores of the tuned and untuned models, so it does seem like tuning it was worthwhile. This is also evident when comparing the top 10 movie recommendations. The tuned RBM model had a decent movie selection, while its counterpart's movies were unrecognizable.

RBMs were a very early type of neural network, and the field of deep learning has evolved considerably since then. A group from the Australian National University published a paper called "AutoRec: Autoencoders Meet Collaborative Filtering". It has three layers: an input layer on the bottom that contains individual ratings, a hidden layer, and an output layer that presents the predictions. A matrix of weights between the layers is maintained across every instance of this network, as well as a bias node for both the hidden and output layers. Learning the weights between the input and hidden layer is called encoding and reconstructing predictions with the weights between the hidden layer and the output layer is called decoding. A sigmoid activation function was used on the output.

All of this is coded in the AutoRec.py file, and like the RBM model, it uses Tensor Flow and is structured in much the same way. Furthermore, AutoRecAlgorithm.py file serves as a wrapper for the AutoRec.py module so it is easier to initialize and use within Frank's core framework.

Just like the RBM model, I could not use RandomizedSearchCV with the AutoRec algorithm. It also took almost double the amount of time, and with GridSearchCV and five-fold cross-validation, Google Colab disconnected my runtime at about 18 hours. I reran the grid search with the most basic hyperparameters and only two-fold cross validation. The results were horrible. Both the tuned and untuned versions of this model

had over double the RMSE of matrix factorization. Hit Rates were worse than Random, and the only decent metric was the Novelty score for the tuned AutoRec model. Qualitatively, the tuned AutoRec model's movie recommendations were actually pretty good, which is surprising considering the horrible RMSE score.

Unfortunately, this particular neural network was unable to outperform matrix factorization on this dataset. Frank says it's partly due to the sparsity issue, and also because 100,000 ratings just isn't anywhere near enough data to train a network as complex as this.

In the end, I would not choose either RBM or AutoRec neural networks over SVD or SVD++. It's not just because the RMSE was higher, but the added complexity and computational time does not warrant the effort. However, because the tuned algorithms had decent movie recommendations, I do experiment with them in some of my Hybrid models.

HYBRID RECOMMENDERS

In the real world, there's no need to choose a single algorithm for your recommender system. Each algorithm has its own strengths and weaknesses and combining many algorithms together could make the sum better than its parts. For example, the winner of the Netflix prize came from a group called KorBell and they won by creating an ensemble of 107 different algorithms!

HybridAlgorithm.py is the last module Frank wrote and it generates rating predictions in parallel and averages their scores together before ranking them. All this code does is iterate through each algorithm, calling estimate (line 25) on each one and combining the results into a weighted average (line 31 - 32), which makes up the final rating estimate (line 34).

Frank says, "Developing recommender systems sometimes feels like more of an art than a science, and choosing how to combine different algorithms can feel like choosing different colors, in a painting, that complement each other." This was definitely the case for me in this section. I experimented with a combination of content-based methods, matrix factorization models, deep learning, and even the Random

algorithm to create 17 different hybrid recommenders. There was added complexity as I had to manually the guess the weights each component algorithm should have.

I often included Surpriselib's Random function, as part of my base algorithm, because I thought it would help add some diversity and "freshness" to the recommendations. However, picking which algorithms to pair it with and how much weight to give it felt like I was throwing darts with a blindfold on.

In the end, I picked three hybrid recommenders to showcase as my best models. Three of them had either SVD or SVD++ as part of its base component which shows how effective matrix factorization is in recommender systems. These three also had low RMSE of about 0.86, similar Hit Rates, and relatively good Novelty scores. All of them had pretty good top 10 movie recommendations, but I choose "Hybrid – 9.4.07" because it had the lowest RMSE and slightly better movie recommendations.

BEST MODEL

Section 10 has a table of my best recommenders from each section. This content-based, deep learning, hybrid, etc. Looking at the metrics alone, I would choose the "Hybrid – 9.4.07" because it had the lowest error, highest Hit Rates, and a decent Novelty score. Interestingly, if I had to choose based solely on the top-10 recommendations, I would have chosen the RBM neural network because it had a better mixture of old and new movies. If I had to pick one, I would choose the hybrid recommender because I believe it to be the more robust of the two when testing online with real users.

RECOMMENDATIONS

If I were to share improvements for my company, this is what I would recommend:

1. Make a cross-functional team to analyze their customers. What are their most popular movies, genres, demographics, etc. I would want to know this because I want to learn the overall Psychology of the customers. In this project, I used top 10 recommendations, but maybe customers only look through

the top five movies on their app before deciding on a movie. This information is important to ensuring the right recommender is being built for the customer (and the company).

2. I know I picked one "best" recommender system, but the ones I've seen in the real world have multiple "top N lists" on their apps and webpages. For example, Netflix has recommendation-lists depending on the last movie/show I watched, the aggregate of my watch history, what's popular for other users, and even one just for new releases. I believe this system is easier to implement and can capture more user engagement rather than just one "best" recommender-list. If I were to use this architecture for my project, I would use my ContentKNN algorithm to recommend popular movies because of the 2018 release years, the Hybrid – 9.4.07 model for recommendations on a user's interests and watch history, and even the Random algorithm to give users the opportunity to find new movies they would love.
3. The third recommendation is to identify the company goal to optimize. Is it recommending movies a user would love to keep them subscribed to the platform longer? Is it recommending certain content sponsored by the platform? These sorts of questions need to be addressed to ensure the recommender system meets the business goal.

AREAS FOR FURTHER EXPLORATION

1. Frank said one of the biggest issues was data sparsity, and admitted that 100k ratings is not enough information. Because of this, some models did not perform as well or as optimal as possible. One avenue I would like to take is to use the full 27 million rating dataset from GroupLens.org.
2. Exploring the full 27 million dataset would probably require me to use a platform like Apache Spark to spread all the data among clusters. Using this platform would be a worthy endeavor in and of itself.
3. The MovieLens dataset has a links.csv which contain movie identifiers for movie databases such as IMDB. By using the identifiers, I can download matching information, for a movie, such as directors, actors, synopses, etc. These added features would allow higher avenues of exploration.

4. I did not use any sort of TF-IDF algorithms as there were no summaries, movie reviews, or synopses to analyze for a recommender system. If I had that information, I could use those as part of an algorithm.
5. Other areas of improvement stems from technical issues. I could only use RandomizedSearchCV with one of my algorithms. With the exception of SVD, I'm not confident in my hyperparameter tuning as it took all day to run and I could only use GridSearchCV. Also, Frank admitted the Hybrid algorithm was rather simple. I had to input the weights of each algorithm and that was how my Hybrid recommender was created. I would like to one day partner with a programmer or do it myself where the weights are learned for each Hybrid model.
6. Instead of an algorithm recommending top ten movies, I would like to customize it by maybe having it only recommend the top eight or nine movies instead. The last two slots could be reserved with random movies or popular movies. Furthermore, Frank's framework did not allow an algorithm to recommend a movie a user has already seen. This meant that favorite movies such as Avengers or Star Wars would never be recommended again. I think allowing a recommendation of at least one watched movie would establish user trust in the recommender system, which could give the user peace-of-mind that the recommendations are relevant for him or her.