Latent Factor Collaborative Filtering for Movie Prediction

Alexander Orlov, Randa Melhem

Abstract
Programmatically predicting someone's tastes, whether applied to music, movies, shopping, or anything else is a very difficult problem that has drawn a lot of attention and funding over the past few years. As soon as companies such as Amazon and Netflix realized what huge implications such a recommender system would have on their businesses they began to funnel money into the research. The Netflix Prize is probably the highest profile example of this - Netflix offered a $1 million reward to any group of researchers that could create a recommendation system that would beat their own by a measure of 10% accuracy of more. It took a couple years, but sure enough someone did.

The method we'll talk about today (collaborative filtering) is what the AT&T researchers used to beat that mark. And while the concept may seem complex at first, collaborative filtering relies mostly on some pretty basic matrix factorization that most people have heard of. There's a lot more that goes into the types of complex models that are used in high-end recommenders systems for large companies, but we'll go over the basic ideas in this paper.

Contents

Introduction

There are two general approaches to recommender systems (oversimplifying the space, but generally). There is a content filtering approach where a profile is created for each user and each product to characterize by nature, and there is a collaborative filtering model, which relies on past user behavior. Content filtering has been successful (see Music Genome Project) in many different areas, however we believe that collaborative filtering will yield a more nuanced and flexible for the type of data that we have.

Types of Collaborative Filtering Collaborative filtering methods can be further broken down into neighborhood methods and latent factor models. Neighborhood models focus on the relationship between users or between items and generally use a simple similarity measure such as Pearson Correlation or euclidean distance. For example, if Netflix were to use a neighborhood model, they would look at my profile and see what I had rated. They might then do a scan over other profiles that are "close" to mine in the space of some similarity measure and then return a weighted average based on distance from my profile. While this is an extremely simple version of a neighborhood model, it illustrates the underlying idea.

Collaborative filtering with a latent factor model allows us to characterize items and users based on some number underlying factors that we choose. For example, if Netflix were using a latent factor model then they might create a recommendation set for me based on which movies were most similar on the range of factors that fit my profile. Instead of using a simple similarity measure for this, the latent factor model would characterize movies based on trained factors such as whether a movie was light or serious, male appealing or female appealing, etc. As you might imagine, this allows for a lot of flexibility and specificity in the model, so it's unsurprising that the team that eventually won the Netflix prize did so using a complex latent factoring model.

1. Models

While we focused mainly on the latent factor model because we thought it would be the strongest, we also used a standard Nearest Neighbor approach.

A tale of two models... It was the best of models, it was the worst of models. Really it was just the worst of models. While we knew where we wanted to go eventually, we decided to start with something rudimentary - a global averages model. Here's the general structure of the model.

$$Y_{um} = \mu + \theta_u + \gamma_m$$

Y_{um} is the average rating that user u gives to a movie m. This is a simplistic model because it only models the simplest possible interaction between users and movies. μ is the global rating mean for a particular movie. θ_u is the user mean that represents the mean rating that a user gives across all movie. In this statistic we hope to capture whether or not a user tends to rate leniently or harshly. Finally, γ_m represents a movie rating average. This attempts to capture the variation in the quality of different movies. For example, Shawshank Redemption is largely regarded as a great film, so we might expect it's average rating to be around 4.5 stars, whereas 50 Shades of Gray was not particularly well-reviewed, so we might expect it to have a rating around 1.5 stars. We added user bias to create a simple differentiation scheme on the assumption that different users may be dramatically different. Therefore the overall mean model was represented as

$$Y_{um} = \mu + (\mu_u - \mu) + (\mu_m - \mu)$$

Neighborhood Distance model We took the idea for a neighborhood distance model from a text called **Programming Collective Intelligence** which has a wonderful introduction to the aforementioned strategy. The model is relatively simple, but still manages to capture a lot of the interaction between user and movie.

Essentially, we choose a certain number of neighbors we want to examine (usually denoted as k as in k-Nearest Neighbors) and then we use a distance metric of our choice to determine which users are closest to the user for whom we're trying to predict. After we find the k nearest users, we rate a given movie for our original user as an average of our k neighbors, weighted by their distance from our original user. Some examples of distance metrics that one could use are Euclidean distance and Pearson Correlation. We personally like Euclidean distance because it has an easily interpretable geometric interpretation, whereas Pearson Correlation Coefficient can be a bit more difficult to glean information from.

Latent Factor Collaborative Filtering The latent factor model characterizes much more complex relationships between users and movies. It uses matrix factorization such that user-item interactions are measure according to the dot product between user vectors and movie vectors. Each user u is represented by a vector p_u and each movie is represented by a vector q_i. Therefore we can capture user-movie interaction by calculating the dot product

$$r_{ui} = q_i^T p_u$$

In vectorized form, the ratings matrix R decomposes into P and Q, where P describes user features and Q describes movie features.

$$R = PQ$$

To learn the factor vectors p_u and q_i we minimize regularized squared error on the set of known movie ratings

$$\min_{q,p} \sum_{(u,i)} \left(r_{ui} - q_i^T p_u\right)^2 + \lambda \left(\|q_i\|^2 + \|p_u\|^2\right)$$

Note that we use regularization to attempt to limit overfitting on the dataset. Lambda is the regularization parameter. The reason that regularization is necessary is that while we train on previously recorded examples of plays, we want to be able to generalize to new users and movie interactions.

There are two main ways to train: stochastic gradient descent and alternating least squares. Stochastic gradient descent operates by iterating over the training set and computing the prediction error. Then the parameters are adjusted in the direction of the negative gradient by some distance η which is chosen by the user. In equation form, at each step

$$e_{ui} = r_{ui} - q_i^T p_u$$

$$q_i \leftarrow q_i + \eta \left(e_{ui} \cdot p_u - \lambda \cdot q_i\right)$$

$$p_u \leftarrow p_u + \eta \left(e_{ui} \cdot q_i - \lambda \cdot p_u\right)$$

Alternating least squares (ALS) is another method by which one can train the interaction vectors. It's also known to be a fast method for matrix factorization (`http://www.jmlr.org/proceedings/papers/v39/kimura14.html`). q_i and p_u are both unknown so at each step in the iteration we hold one fixed and solve the least squares approximation for the other. There are tradeoffs between stochastic gradient descent and ALS, but in the end we decided to use stochastic gradient descent.

Ideally we could have added biases into the model for the same reasons that we discussed earlier (in order to differentiate between different users and artists more accurately), however we didn't have enough time to do so. In this model we'll represent bias as

$$b_{ui} = \mu + b_i + b_u$$

Therefore we add in the bias to yield a predicted rating for a user u and a movie i

$$r_{ui} = \mu + b_i + b_u + q_i^T p_u$$

Note that biases can be negative because they're calculated relative to the global mean for both users and movies. The new expression for prediction is therefore

$$\min_{p,q,b} \sum_{(u,i)} \left(r_{ui} - \mu - b_u - b_i - p_u^T q_i\right)^2 + \lambda \left(\|p_u\|^2 + \|q_i\|^2 + b_u^2 + b_i^2\right)$$

Where we ended up Here's where we finally ended up with the Latent Factor model. We implemented the matrix decomposition via SVD to create the matrices q and p. After a lot of debugging we finally gave up on attempting to implement the gradient descent method. Right now the model creates a low-rank approximation of p and q to minimize reconstruction error. The only fallback here is that without being able to train using gradient descent, the model is largely prone to overfitting. That happens mainly for 2 reasons. Firstly, traditional SVD is undefined where there are missing values in a matrix. We represent missing values as 0, however that's not technically correct. There are many approaches that exist in the literature that revolve around carefully constructing averages to fill in missing values, however we chose to not sully the integrity of the data by adding in our own bias, as we worried about what that might do to the model. Secondly, if we reconstruct based simply on the low-rank approximation then we capture most of the ”essence” of the data, but at the end of the day what we’re really doing is simply reconstructing the original data matrix, meaning that we’re largely overfitting to the training data. Stochastic gradient descent would have allowed us to avoid this, but unfortunately we couldn’t get the darn algorithm working. The code for the stochastic gradient descent remains in the latent factor model (a vectorized and non-vectorized version) so you can take a look if you’re curious.