Capstone Project 1: In Depth Analysis

Joseph Tran

Springboard 2020

This report summarizes the methods used to implement an implicit recommender system. In the analysis the implicit and lightfm libraries are used to implement collaborative filtering using matrix factorization. The implicit library implements alternating least squares to decompose the user/item matrix into latent user/item vectors used for predictions. The lightfm library offers the flexibility of creating a hybrid model, and offers several loss functions for the collaborative filtering algorithm. A model is implemented in the implicit library and the auc score for the predictions are compared to popular recommendations. Another model is implemented in lightfm, and the auc score is then calculated and compared with the implicit model.

Explicit v. Implicit Data

The data for recommendation systems comes in two flavors: explicit and implicit. Explicit data is user defined data. An example of explicit data would be a user supplied rating based on some type of scale. This type of data relies on user input to rate an item, and as a result much less is generated. At the same time, ratings will depend on the generosity of the user. A five star rating may mean something different to a professional critic, than it would for the average user.

Implicit data does not rely on user input, and is generated from a user interacting with the platform. As a result, the data is more widely available. The implicit data that we have for our model are listening events. The listening events can be used to infer whether a user liked or disliked a song. We can introduce a notion of confidence by looking at how many times a user interacted with an item. For example, if a user listened to a song 16 times, we can have more confidence that the user liked the song than if the user had only listened to it once.

Creating User/Item Matrix

In order to compute the latent user/item features using matrix factorization we need to create a user/item matrix where the rows are users, and columns are items. With smaller datasets, this can be achieved with the pivot function in pandas, and stored as a dataframe. However, user/item matrices are generally quite large and very sparse. This leads to memory errors when trying to perform the pivot in pandas. A better structure for the user item matrix is a compressed sparse row, or csr matrix. Not only is the csr matrix required to store the data, both libraries require that the input be in the form of a csr matrix. The data is loaded with the pandas read csv function loading only the user id and track id. A count column is created and the value is set to 1. This is so we can get a count of how many times a user listened to a song. Next, we group

by user id and track id and aggregate the values in the count column by summing. This leaves us with a dataframe that contains the user id, track id, and the count of interactions per user and item as a confidence metric. Lists of unique user/item id's are created to specify the shape of the matrix, and category codes for each user/item are created for sequential indexing. The csr matrix is created by supplying the lists to the function and specifying its shape. In our case, we generate a csr matrix with a shape of 21220 users and 81343 items, with 1,053,887 stored elements representing user/item interactions.

Creating Train/Test Split

In the standard supervised machine learning model, splitting the data into train and test sets is fairly straightforward. One can simply sample a small percentage of rows in the data for testing, and then train on the remaining data. This method is inadequate for our model because we need the interactions to complete the matrix factorization. An alternate method is to make two copies of the user/item matrix, alter a percentage of the non zero interactions and use the altered user/item matrix to perform the matrix factorization during training. The test set is then stored as a binary preference matrix with the interactions set to 1. This is achieved by making copies of the data and using the train set copy to find the indices where interactions took place. These indices are then stored and sampled from, for interactions to mask. We then re-assign the value of zero for the sampled interactions indices in the training set.

Implicit: Alternating Least Squares

The alternating least squares algorithm was chosen for its ability to introduce a confidence value to each interaction. As mentioned before, we can use the amount of times a user listened to a song to put a confidence on how well we believe a user liked an item. When the data is implicit, the goal of the model is to be able to say will the user like the item. This is represented by 0 and 1 with 1 meaning they will like it, and 0 denoting no preference.

$$p_{ui} \, \varepsilon \, (0, 1)$$

It is important to note that a value of 0 does not mean a user will not like an item, which means a user is likely to like it, rather than a user is likely to dislike it. As such, this model will not be able to tell us what a user does not like. The next component in the model is the recording. In the case of explicit data, this would be the rating a user gave to an item. In implicit data this represents the recording of how many times a user interacted with an item, or in this case, listened to a song. These are the values in our user item matrix.

$$r_{ui} \, \varepsilon \, R$$

The relationship between the preference and recording is as follows…

$$p_{ui} = \{1 \; if \; r_{ui} > 0, \; 0 \; if \; r_{ui} = 0\}$$

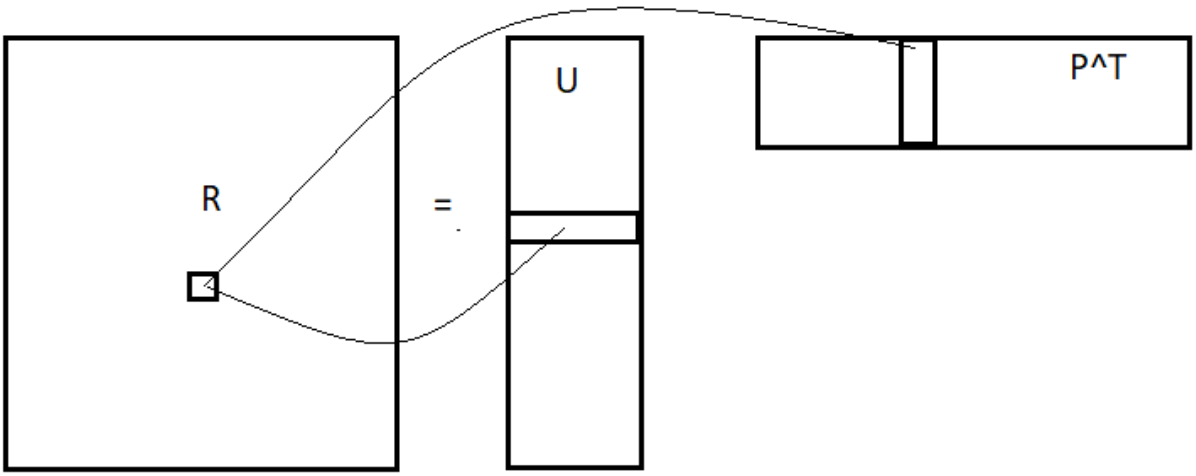
In the case of our listening events, the equation states that if a user listened to a song once, they liked the song, otherwise no preference. However, listening to a song once does not necessarily mean that the song was liked. This is where the notion of confidence is introduced.

$$C_{ui} = 1 + \alpha r_{ui}$$

This above confidence equation allows us to have some confidence in listening events that occur one or a few times, and more confidence in interactions that took place many times. At the same time we would like to have some confidence that an interaction is 0. If r_{ui} is zero, the confidence that there is no preference in the item is captured in the 1 out front. α is a tuning parameter that weights the confidence that a listening event is 0 or 1. When putting this all together to implement alternating least squares, the equation we want to minimize is...

$$C_{ui}(p_{ui} - U_u X_i^T)$$

Where U_u is the user vector, and X_i^T is the item vector. The goal is to find the user vector U, and item vector X, such that the equation is minimized.



The above image is a representation of the task at hand. We start with a sparse user/item matrix, R, and perform alternating least squares to approximate the latent vectors U and P^T. The algorithm achieves this by holding U constant,and solving for P^T, and then holding P^T constant and solving for U. The algorithm alternates between the two until convergence is found. The shape of U and P^T is determined in part by the shape of the ratings matrix. The latent user vector will be the length of unique users, and the item vector the length of the number of unique songs. The width of each is defined by the number of components, or latent factors chosen when instantiating the model. For our model the number of components was chosen to be 20. The method of determining the number of components by examining explained variance was attempted, however the local system did not have enough memory to run the routine.

The model is created by instantiating the alternating least squares algorithm in the implicit library with the factors set to 20. The alpha parameter is set to 40, as it is mentioned in the literature that 40 is a good value to start with. The output of the model are the latent user and item feature vectors that are used to make predictions. The shape of each is checked to ensure the shapes are as expected.

In order to evaluate the model, we will use the predictions at the indices where interactions were masked, and compare these values to the binarized test set. We want to examine how the order of recommendations for each user with a masked interaction compares to the songs they actually listened to. For this, we will use the ROC, or receiver operating curve. A greater area under the curve means the user listened to songs near the top of the recommendation list. In order to do this a function that calculates area under the curve is implemented for all users that had interactions masked in the training set. An AUC score is also calculated for popular items to see how our recommendations compare to just recommending the most popular items. After running the function, we are left with AUC scores of 0.916 and 0.875 for the ALS algorithm and popular recommendations, respectively. The ALS algorithm beat the popularity measure by about 4%, this is encouraging as it is difficult to beat popularity recommendation models. At this point, we could try to nudge the parameters, or increase the number of components, but a better way to increase the score would be to add more data. This is where the lighfm library comes in handy.

Lightfm Model

The lightfm library was chosen for its capability to construct hybrid models. It has the ability to add user and item features to enhance the quality of the recommendations by taking into account user and item metadata such as, hashtags, artist name, song features, and any other user or item metadata. The hybrid model is beyond the scope of this project, but the lightfm library was chosen in the event that the project is extended. In our case, if no user or item features are supplied to the model, the algorithm reduces to the standard matrix factorization.

This standard model is used to compare the performance of the Implicit model. To use the lightfm model, an instance needs to be instantiated with the number of components and loss function. For fair comparison, we choose 20 components to be in line with the ALS model. There are several options to choose for the loss function. The loss functions that are relevant to the data we have are the 'BPR,' Bayesian Personalized Ranking and 'WARP,' weighted approximate rank pairwise. The 'BPR' method maximizes between a positive example and a randomly chosen negative example, and is useful when only positive interactions are present and optimization for ROC AUC is desired. The 'WARP' method maximizes the rank of positive examples by repeatedly sampling negative examples until a rank violating one is found, and is useful when only positive interactions are found and optimising the top of the recommendation list is desired.

Next, use lightfm's dataset class to fit the data. This is done by instantiating the dataset class and supplying the unique users and items. The model instance is then fit with the train data. Afterwards, we score the model using lightfm's AUC function using the test data.

Both the 'BPR' and 'WARP' loss functions are used to see which produces a better score. The AUC using WARP was found to be 93.7%, which beats both the ALS model and the popularity. The model using the 'BPR' loss function was scored and found to be 80.6%, losing out to the ALS, WARP, and popularity models. This was somewhat confusing as the BPR loss function is used when trying to optimize AUC.

Conclusion

At this point, we could try and improve the results by creating a hybrid model within lightfm. This would allow for the use of the item data contained in the nowplaying_rs data set. However, this is beyond the scope of this work and will be looked at in the future. Both the Implicit and lightfm models scored higher than popular recommendations, with the lightfm model beating ALS by a few percentage points, this is determined to be sufficient for the purposes of building a basic recommendation system with matrix factorization.