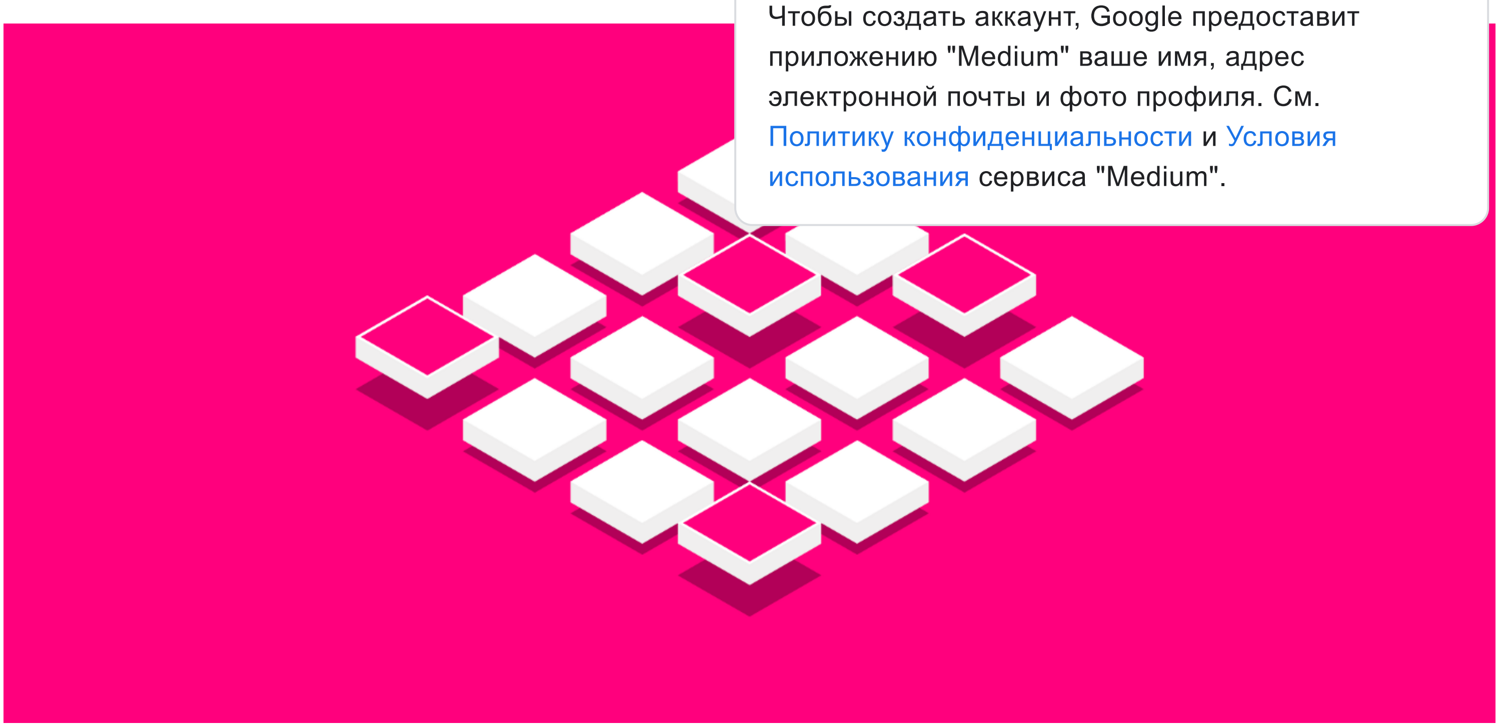
Published in Rn Engineering · Follow

Victor · Follow
Aug 23, 2017 · 12 min read



ALS Implicit Collaborative Filtering

Continuing on the collaborative filtering theme from my collaborative filtering with binary data example i'm going to look at another way to do collaborative filtering using matrix factorization with implicit data.

This story relies heavily on the work of Yifan Hu, Yehuda Koren, Chris Volinsky in their paper on Collaborative Filtering for Implicit Feedback as well as code and concepts from Ben Frederickson, Chris Johnson, Jesse Steinweg-Woods and Erik Bernhardsson.

Content:

- **Overview**
- **Implicit vs explicit**
- **The dataset**
- **Alternating least squares**

- Ok, let's write it! (the code)
- Summary
- References

Overview

We're going to write a simple implementation o recommendation algorithm. We want to be able recommendations for our users. I will focus on b different python implementations.

Since we're taking a collaborative filtering approach we will only be concern ourselves with items, users and what items a user has interacted with.

Implicit vs explicit data

Explicit data is data where we have some sort of rating. Like the 1 to 5 ratings from the MovieLens or Netflix dataset. Here we know how much a user likes or dislikes an item which is great, but this data is hard to come by. Your users might not spend the time to rate items or your app might not work well with a rating approach in the first place.

Implicit data (the type of data we're using here) is data we gather from the users behaviour, with no ratings or specific actions needed. It could be what items a user purchased, how many times they played a song or watched a movie, how long they've spent reading a specific article etc. The upside is that we have a lot more of this data, the downside is that it's more noisy and not always apparent what it means.

For example, with star ratings we know that a **1** means the user did not like that item and a **5** that they really loved it. With song plays it might be that the user played a song and hated it, or loved it, or somewhere in-between. If they did not play a song it might be since they don't like it or that they would love it if they just knew about.

So instead we focus on what we know the user has consumed and the *confidence* we have in whether or not they like any given item. We can for example measure how often they play a song and assume a higher confidence if they've listened to it 500 times vs. one time.

Implicit recommendations are becoming an increasingly important part of many recommendation systems as the amount of implicit data grows. For example the original Netflix challenge focused only on explicit data but they're now relying more and more on implicit users. It contains the user id, an artist id, the na

played any given artist. The download also cont

etc. but we'll not be using that now.

Alternating Least Squares

Alternating Least Squares (ALS) is a the model v

before we dive into how it works we should look

which is what we aim to use ALS to accomplish.

Matrix factorization

The idea is basically to take a large (or potentially huge) matrix and factor it into some smaller representation of the original matrix. You can think of it in the same way as we would take a large number and factor it into two much smaller primes. We end up with two or more lower dimensional matrices whose product equals the original one.

When we talk about collaborative filtering for recommender systems we want to solve the problem of our original matrix having millions of different dimensions, but our "tastes" not being nearly as complex. Even if i've viewed hundreds of items they might just express a couple of different tastes. Here we can actually use matrix factorization to mathematically reduce the dimensionality of our original "all users by all items" matrix into something much smaller that represents "all items by some taste dimensions" and "all users by some taste dimensions". These dimensions are called ***latent or hidden features*** and we learn them from our data.

Doing this reduction and working with fewer dimensions makes it both much more computationally efficient and but also gives us better results since we can reason about items in this more compact "taste space".

If we can express each user as a vector of their taste values, and at the same time express each item as a vector of what tastes they represent. You can see we can quite easily make a recommendation. This also gives us the ability to find connections between users who have no specific items in common but share common tastes.

If we can express each user as a vector of their taste values, and at the same time express each item as a vector of what tastes they represent. You can see we can quite easily make a recommendation

any real metadata.

Matrix factorization Implicit data

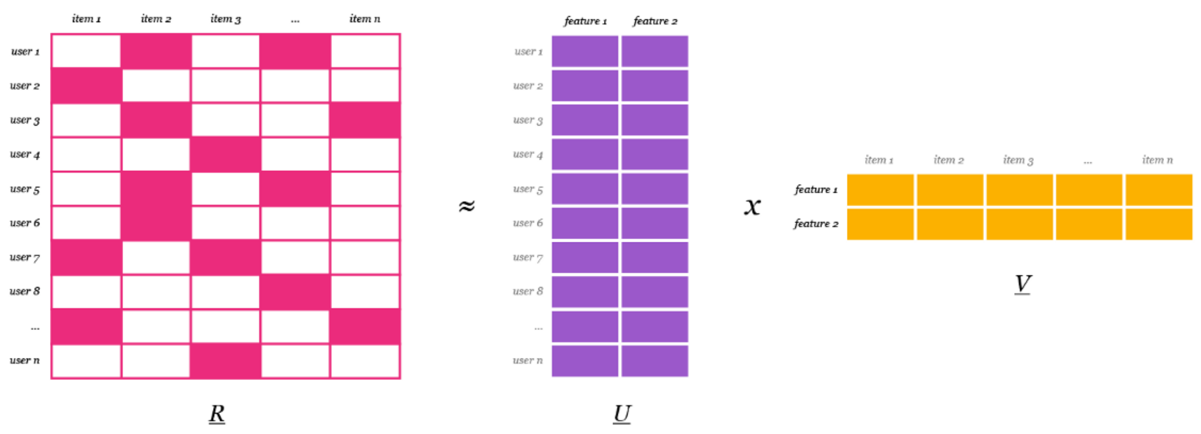
There are different ways to factor a matrix, like
Probabilistic Latent Semantic Analysis (PLSA) if

With implicit data the difference lies in how we
matrix. For explicit data we treat them as just u
predicted rating to. But for implicit we can't just
these unknown values as well. As stated before
disliked something, or if it means they love it but just don't know about it. Basically we need some way to *learn* from the missing data. So we'll need a different approach to get us there.

Back to ALS

ALS is an iterative optimization process where we for every iteration try to arrive closer and closer to a factorized representation of our original data.

We have our original matrix ***R*** of size *u x i* with our users, items and some type of feedback data. We then want to find a way to turn that into one matrix with users and hidden features of size *u x f* and one with items and hidden features of size *f x i*. In ***U*** and ***V*** we have weights for how each user/item relates to each feature. What we do is we calculate ***U*** and ***V*** so that their product approximates ***R*** as closely as possible: ***R*** ≈ ***U*** **x** ***V***.



By randomly assigning the values in ***U*** and ***V*** and using *least squares* iteratively we can arrive at what weights yield the best approximation of ***R***. The *least squares* approach in it's basic forms

R = U x V.

The approach we're going to use with our implic *Filtering for Implicit Feedback Datasets* by Hu and Spotify). Their solution is very straight forw and implementation but you should definitely g

Their solution is to merge the *preference (p)* for preference. We start out with missing values as value and existing values a positive preference b something like play count, time spent on a page or some other form of interaction as the basis for calculating our confidence.

- **We set the preference (*p*):**

$$p_{ui} = \begin{cases} 1 & r_{ui} > 0 \\ 0 & r_{ui} = 0 \end{cases}$$

Basically our preference is a binary representation of our feedback data **r**. If the feedback is greater than zero we set it to 1. Make sense.

- **The confidence (*c*) is calculated as follows:**

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

Here the confidence is calculated using the magnitude of **r** (the feedback data) giving us a larger confidence the more times a user has played, viewed or clicked an item. The rate of which our confidence increases is set through a linear scaling factor **α**. We also add 1 so we have a minimal confidence even if **α x r** equals zero.

This also means that even if we only have one interaction between a user and item the confidence will be higher than that of the unknown data given the **α** value. In the paper they

Войдите в приложение "Medium" с аккаунтом Google

Михаил Соколов
sokolovmisha44@gmail.com

Продолжить как Михаил

Чтобы создать аккаунт, Google предоставит приложению "Medium" ваше имя, адрес электронной почты и фото профиля. См. Политику конфиденциальности и Условия использования сервиса "Medium".

$$\min_{y_*, y_*} \sum_{u,i} c_{ui}(p_{ui} - x_u^T y_i)^2 +$$

As the paper notes, if we fix the user factors or it
The derivative of the above equation gets us the following equation for minimizing the loss of our users:

$$x_u = (Y^T C^u Y + \lambda I)^{-1} Y^T C^u p(u)$$

And the this for minimizing it for our items:

$$y_i = (X^T C^i X + \lambda I)^{-1} X^T C^i p(i)$$

One more step is that by realizing that the product of ***Y-transpose***, ***Cu*** and ***Y*** can be broken out as shown below:

$$Y^T C_u Y = Y^T Y + Y^T (C_u - I) Y$$

Now we have ***Y-transpose-Y*** and ***X-transpose-X*** independent of ***u*** and ***i*** which means we can precompute it and make the calculation much less intensive. So with that in mind our final user and item equations are:

$$x_u = (Y^T Y + Y^T (C^u$$

$$y_i = (X^T X + X^T (C^i$$

- ***X and Y:*** *Our randomly initialized user and it*
- ***Cu and Ci:*** *Our confidence values.*
- ***λ:*** *Regularizer to reduce overfitting (we're using 0.1).*
- ***p(u) and p(i):*** *The binary preference for an item. One if we know the preference and zero if we don't.*
- ***I (eye):*** *The identity matrix. Square matrix with ones on the diagonal and zeros everywhere else.*

By iterating between computing the two equations above we arrive at one matrix with user vectors and one with item vectors that we can then use to produce recommendations or find similarities.

Similar items

To calculate the similarity between items we compute the dot-product between our item vectors and it's transpose. So if we want artists similar to say Joy Division we take the dot product between all item vectors and the transpose of the Joy Division item vector. This will give us the similarity score:

$$score = V \cdot V_i^T$$

Making recommendations

To make recommendations for a given user we take a similar approach. Here we calculate the dot product between our user vector and the transpose of our item vectors. This gives us a recommendation score for our user and each item:

score

Войдите в приложение "Medium" с аккаунтом Google

Михаил Соколов
sokolovmisha44@gmail.com

Продолжить как Михаил

Чтобы создать аккаунт, Google предоставит приложению "Medium" ваше имя, адрес электронной почты и фото профиля. См. Политику конфиденциальности и Условия использования сервиса "Medium".

Ok, let's write it!

First we'll import the libraries we need and load
wrangling to get the data into the shape we war

We then create our sparse matrix ***R*** (data_sparse) of size *users x items*. Using a sparse matrix

Now when we have our data prepped and ready
our implicit ALS function.

We start out by calculating the confidence for al
hold our user and item vectors and randomly as
diagonals.

Войдите в приложение "Medium" с аккаунтом Google

Михаил Соколов
sokolovmisha44@gmail.com

Продолжить как Михаил

Чтобы создать аккаунт, Google предоставит приложению "Medium" ваше имя, адрес электронной почты и фото профиля. См. Политику конфиденциальности и Условия использования сервиса "Medium".

Still inside our implicit_als function we start the main iteration loop. Here we first precompute X-transpose-X and Y-transpose-Y as discussed earlier. We then have two inner loops where we first

We then call our function to get our user vectors and item vectors. As you'll notice if you try to run this with the dataset we're using it will take a VERY long time to train. We'll get back to how we can speed it up later but for now we can try with just one iteration instead of 20. Or we can slice the raw data into something more manageable, say just the first 100,000 rows.

So now when we have our trained model we can start making some recommendations. First let’s start by just finding some artists similar Jay-Z. We get the similarity by talking the dot product of our item vectors with the item vector of the artist.

Running on a small subset of the dataset i get that the most similar artists in order to Jay-Z are: Jay-Z, 50 Cent, Kanye West, Eminem, NAS and Norah Jones, Akon and 2Pac. Looks like a fairly good prediction.

Now let’s generate some recommendations for a user. Here most of the code is just moving, reshaping and making the results readable. To get the actual score we take the dot product between the trained user vector and the transpose of the item vectors.

Another notable part is the MinMaxScaler where we take our recommendation scores and scale them within a 0 to 1 range. This does not change the result but makes things a bit neater.

Speeding it up

As mentioned above if you try to run the above implementation of our implicit_als function with 20 iterations on the full lastfm 360K dataset it will take a VERY long time. I have no idea how long since i got bored waiting for the first iteration to complete. Using only 100K rows of the data 20 iterations took about 30 minutes.

Following the implementation and code by Ben Frederickson we can replace our implicit_als function with the below code and speed things up quite a bit. Here we're using the approach outlined in this paper using the Conjugate Gradient (CG) method.

Running this for 20 iteration on the same 100K

Even faster implicit with implicit!

Ben Frederickson also has a super nice Cython implementation that you should definitely use. This combines the above speedup with the performance of C. We will have to tweak our code slightly to fit with his library, the main difference being it expects training data of shape *items x users*.

Now we can run the same 100K rows in just 0.4
minute.

Implicit also has built in functions for recommen
down to just a few source lines of code:

References and resources:

- http://yifanhu.net/PUB/cf.pdf
- https://jessesw.com/Rec-System/
- https://github.com/benfred/implicit
- http://www.benfrederickson.com/matrix-f
- http://www.benfrederickson.com/fast-imp
- https://codeascraft.com/2014/11/17/personalized-recommendations-at-etsy/
- https://pdfs.semanticscholar.org/bfdf/7af6cf7fd7bb5e6b6db5bbd91be11597eaf0.pdf
- https://vimeo.com/57900625
- https://github.com/MrChrisJohnson/implicit-mf