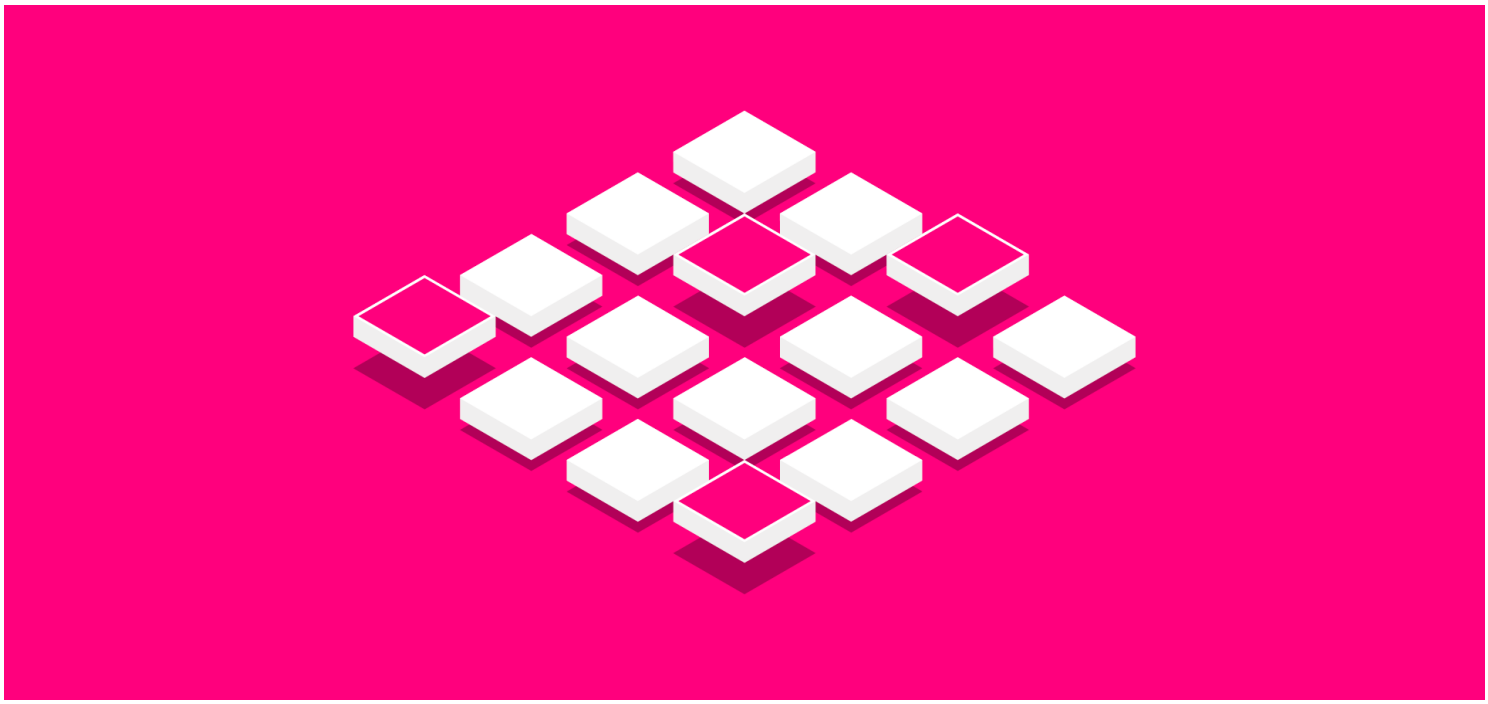


ALS Implicit Collaborative Filtering

Victor Follow
Aug 24, 2017 · 12 min read

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**

Overview

We're going to write a simple implementation of an implicit (more on that below) recommendation algorithm. We want to be able to find similar items and make recommendations for our users. I will focus on both the theory, some math as well as a couple of 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 signals. The same thing goes for Hulu, Spotify, Etsy and many others.

The dataset

For this example we'll be using the lastfm dataset containing the listening behaviour of 360,000 users. It contains the user id, an artist id, the name of the artists and the number of times a user played any given artist. The download also contains a file with user ages, geners and countries etc. but we'll not be using that now.

Alternating Least Squares

Alternating Least Squares (ALS) is a the model we'll use to fit our data and find similarities. But before we dive into how it works we should look at some of the basics of *matrix factorization* 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.

Now it should be noted that we have no idea of what these features or tastes really are. We won’t be able to label them “rock” or “fast paced” or “featuring Jay-Z”. They don’t necessarily reflect any real metadata.

Matrix factorization Implicit data

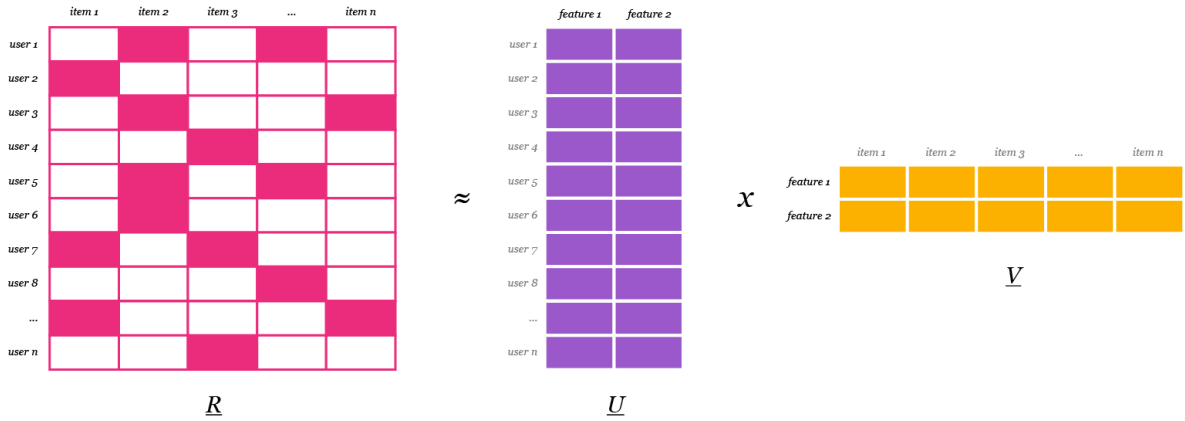
There are different ways to factor a matrix, like Singular Value Decomposition (SVD) or Probabilistic Latent Semantic Analysis (PLSA) if we’re dealing with explicit data.

With implicit data the difference lies in how we deal with all the missing data in our very sparse matrix. For explicit data we treat them as just unknown fields that we should assign some predicted rating to. But for implicit we can’t just assume the same since there is information in these unknown values as well. As stated before we don’t know if a missing value means the user 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: $R \approx U \times V$.



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 means fitting some line to the data, measuring the sum of squared distances from all points to the line and trying to get an optimal fit by minimising this value.

With the *alternating least squares* approach we use the same idea but iteratively alternate between optimizing **U** and fixing **V** and vice versa. We do this for each iteration to arrive closer to $R = U \times V$.

The approach we're going to use with our implicit dataset is the one outlined in ***Collaborative Filtering for Implicit Feedback Datasets*** by Hu, Korenand and Volinsky (and used by Facebook and Spotify). Their solution is very straight forward so i'm just

going to explain the general idea and implementation but you should definitely give it a read.

Their solution is to merge the *preference (**p**)* for an item with the *confidence (**c**)* we have for that preference. We start out with missing values as a negative preference with a low confidence value and existing values a positive preference but with a high confidence value. We can use 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 found $\alpha = 40$ to work well and somewhere between 15 and 40 worked for me.

The goal now is to find the vector for each user (**xu)** and item **(yi)** in feature dimensions which means we want to minimize the following loss function:

$$\min_{y_*, y_*} \sum_{u,i} c_{ui}(p_{ui} - x_u^T y_i)^2 + \lambda \left(\sum_u \| x_u \|^2 + \sum_i \| y_i \|^2 \right)$$

As the paper notes, if we fix the user factors or item factors we can calculate a global minimum. 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:

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

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

- ***X and Y:*** *Our randomly initialized user and item matricies. These will get alternatingly updated.*
- ***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:

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

$$score = V \cdot V_i$$

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:

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

Ok, let's write it!

First we'll import the libraries we need and load the lastfm dataset. We will also need to do some wrangling to get the data into the shape we want.

```
import random
import pandas as pd
import numpy as np

import scipy.sparse as sparse
from scipy.sparse.linalg import spsolve
from sklearn.preprocessing import MinMaxScaler

#------------------------
# LOAD AND PREP THE DATA
#------------------------

raw_data = pd.read_table('data/usersha1-artmbid-artname-plays.tsv')
raw_data = raw_data.drop(raw_data.columns[1], axis=1)
raw_data.columns = ['user', 'artist', 'plays']

 # Drop rows with missing values
 data = raw_data.dropna()

 # Convert artists names into numerical IDs
 data['user_id'] = data['user'].astype("category").cat.codes
 data['artist_id'] = data['artist'].astype("category").cat.codes
```

```
23
24  # Create a lookup frame so we can get the artist names back in
25  # readable form later.
26  item_lookup = data[['artist_id', 'artist']].drop_duplicates()
27  item_lookup['artist_id'] = item_lookup.artist_id.astype(str)
28
29  data = data.drop(['user', 'artist'], axis=1)
30
31  # Drop any rows that have 0 plays
32  data = data.loc[data.plays != 0]
33
34  # Create lists of all users, artists and plays
35  users = list(np.sort(data.user_id.unique()))
36  artists = list(np.sort(data.artist_id.unique()))
37  plays = list(data.plays)
38
39  # Get the rows and columns for our new matrix
40  rows = data.user_id.astype(int)
41  cols = data.artist_id.astype(int)
42
43  # Contruct a sparse matrix for our users and items containing number of plays
44  data_sparse = sparse.csr_matrix((plays, (rows, cols)), shape=(len(users), len(artists)))
```

We then create our sparse matrix R (data_sparse) of size *users x items*. Using a sparse matrix allows us to store only the values that are actually there and not all the missing ones (which is about 99% of the dataset.)

Now when we have our data prepped and ready to go we can start on a first implementation of our implicit ALS function.

We start out by calculating the confidence for all users and items, create our X and Y matrices to hold our user and item vectors and randomly assign the values. We also precompute our I diagonals.

```
1
2  def implicit_als(sparse_data, alpha_val=40, iterations=10, lambda_val=0.1, features=10):
3
4  """ Implementation of Alternating Least Squares with implicit data. We iteratively
5      compute the user (x_u) and item (y_i) vectors using the following formulas:
6
7      x_u = ((Y.T*Y + Y.T*(Cu - I) * Y) + lambda*I)^-1 * (X.T * Cu * p(u))
8      y_i = ((X.T*X + X.T*(Ci - I) * X) + lambda*I)^-1 * (Y.T * Ci * p(i))
```

```
    Args:
        sparse_data (csr_matrix): Our sparse user-by-item matrix

        alpha_val (int): The rate in which we'll increase our confidence
        in a preference with more interactions.

        iterations (int): How many times we alternate between fixing and
        updating our user and item vectors

        lambda_val (float): Regularization value

        features (int): How many latent features we want to compute.

    Returns:
        X (csr_matrix): user vectors of size users-by-features

        Y (csr_matrix): item vectors of size items-by-features
     """

    # Calculate the foncidence for each value in our data
    confidence = sparse_data * alpha_val

    # Get the size of user rows and item columns
    user_size, item_size = sparse_data.shape

    # We create the user vectors X of size users-by-features, the item vectors
    # Y of size items-by-features and randomly assign the values.
    X = sparse.csr_matrix(np.random.normal(size = (user_size, features)))
    Y = sparse.csr_matrix(np.random.normal(size = (item_size, features)))

    #Precompute I and lambda * I
    X_I = sparse.eye(user_size)
    Y_I = sparse.eye(item_size)

    I = sparse.eye(features)
    lI = lambda_val * I
```

implicit_feedback_als_2.py hosted with ♡ by GitHub view raw

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 iterate over all users and update X and then do the same for all items and update Y.

All we have to do to calculate X[u] and Y[i] is implement the formula from the paper above.

```
""" Continuation of implicit_als function"""

    # Start main loop. For each iteration we first compute X and then Y
    for i in xrange(iterations):
        print 'iteration %d of %d' % (i+1, iterations)

        # Precompute Y-transpose-Y and X-transpose-X
        yTy = Y.T.dot(Y)
        xTx = X.T.dot(X)

        # Loop through all users
        for u in xrange(user_size):

            # Get the user row.
            u_row = confidence[u,:].toarray()

            # Calculate the binary preference p(u)
            p_u = u_row.copy()
            p_u[p_u != 0] = 1.0

            # Calculate Cu and Cu - I
            CuI = sparse.diags(u_row, [0])
            Cu = CuI + Y_I

            # Put it all together and compute the final formula
            yT_CuI_y = Y.T.dot(CuI).dot(Y)
            yT_Cu_pu = Y.T.dot(Cu).dot(p_u.T)
            X[u] = spsolve(yTy + yT_CuI_y + lI, yT_Cu_pu)


        for i in xrange(item_size):

            # Get the item column and transpose it.
            i_row = confidence[:,i].T.toarray()

            # Calculate the binary preference p(i)
            p_i = i_row.copy()
            p_i[p_i != 0] = 1.0

            # Calculate Ci and Ci - I
```

```
            # Calculate Ci and Ci - I
            CiI = sparse.diags(i_row, [0])
            Ci = CiI + X_I

            # Put it all together and compute the final formula
            xT_CiI_x = X.T.dot(CiI).dot(X)
            xT_Ci_pi = X.T.dot(Ci).dot(p_i.T)
            Y[i] = spsolve(xTx + xT_CiI_x + lI, xT_Ci_pi)

    return X, Y
```

implicit_feedback_als_3.py hosted with ♡ by GitHub view raw

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.

```

user_vecs, item_vecs = implicit_als(data_sparse, iterations=20, features=20, alpha_val=40)
```

implicit_feedback_als_4.py hosted with ♡ by GitHub view raw

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.

```

#------------------------------
# FIND SIMILAR ITEMS
#------------------------------

# Let's find similar artists to Jay-Z.
# Note that this ID might be different for you if you're using
# the full dataset or if you've sliced it somehow.
item_id = 10277

# Get the item row for Jay-Z
item_vec = item_vecs[item_id].T

# Calculate the similarity score between Mr Carter and other artists
# and select the top 10 most similar
```

```
# and select the top 10 most similar.
scores = item_vecs.dot(item_vec).toarray().reshape(1,-1)[0]
top_10 = np.argsort(scores)[::-1][:10]

artists = []
artist_scores = []

# Get and print the actual artists names and scores
for idx in top_10:
    artists.append(item_lookup.artist.loc[item_lookup.artist_id == str(idx)].iloc[0])
    artist_scores.append(scores[idx])

similar = pd.DataFrame({'artist': artists, 'score': artist_scores})

print similar
```

implicit_feedback_als_6.py hosted with ♡ by GitHub view raw

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.

```

# Let's say we want to recommend artists for user with ID 2023
user_id = 2023

#------------------------------
# GET ITEMS CONSUMED BY USER
#------------------------------

# Let's print out what the user has listened to
consumed_idx = data_sparse[user_id,:].nonzero()[1].astype(str)
consumed_items = item_lookup.loc[item_lookup.artist_id.isin(consumed_idx)]
print consumed_items

```

```
14
15  #------------------------------
16  # CREATE USER RECOMMENDATIONS
17  #------------------------------
18
19  def recommend(user_id, data_sparse, user_vecs, item_vecs, item_lookup, num_items=10):
20      """Recommend items for a given user given a trained model
21
22      Args:
23          user_id (int): The id of the user we want to create recommendations for.
24
25          data_sparse (csr_matrix): Our original training data.
26
27          user_vecs (csr_matrix): The trained user x features vectors
28
29          item_vecs (csr_matrix): The trained item x features vectors
30
31          item_lookup (pandas.DataFrame): Used to map artist ids to artist names
32
33          num_items (int): How many recommendations we want to return:
34
35      Returns:
36          recommendations (pandas.DataFrame): DataFrame with num_items artist names and scores
37
38      """
39
40      # Get all interactions by the user
41      user_interactions = data_sparse[user_id,:].toarray()
42
43      # We don't want to recommend items the user has consumed. So let's
44      # set them all to 0 and the unknowns to 1.
45      user_interactions = user_interactions.reshape(-1) + 1 #Reshape to turn into 1D array
46      user_interactions[user_interactions > 1] = 0
47
48      # This is where we calculate the recommendation by taking the
49      # dot-product of the user vectors with the item vectors.
50      rec_vector = user_vecs[user_id,:].dot(item_vecs.T).toarray()
51
52      # Let's scale our scores between 0 and 1 to make it all easier to interpret.
53      min_max = MinMaxScaler()
54      rec_vector_scaled = min_max.fit_transform(rec_vector.reshape(-1,1))[:,0]
55      recommend_vector = user_interactions*rec_vector_scaled
56
57      # Get all the artist indices in order of recommendations (descending) and
58      # select only the top "num_items" items.
```

```
    item_idx = np.argsort(recommend_vector)[::-1][:num_items]

    artists = []
    scores = []

    # Loop through our recommended artist indicies and look up the actial artist name
    for idx in item_idx:
        artists.append(item_lookup.artist.loc[item_lookup.artist_id == str(idx)].iloc[0])
        scores.append(recommend_vector[idx])

    # Create a new dataframe with recommended artist names and scores
    recommendations = pd.DataFrame({'artist': artists, 'score': scores})

    return recommendations

# Let's generate and print our recommendations
recommendations = recommend(user_id, data_sparse, user_vecs, item_vecs, item_lookup)
print recommendations
```

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.

```

def nonzeros(m, row):
    for index in xrange(m.indptr[row], m.indptr[row+1]):
        yield m.indices[index], m.data[index]


def implicit_als_cg(Cui, features=20, iterations=20, lambda_val=0.1):
    user_size, item_size = Cui.shape

    X = np.random.rand(user_size, features) * 0.01
    Y = np.random.rand(item_size, features) * 0.01

    Cui, Ciu = Cui.tocsr(), Cui.T.tocsr()
```

```
    Cui, Ciu = Cui.tocsr(), Cui.T.tocsr()

    for iteration in xrange(iterations):
        print 'iteration %d of %d' % (iteration+1, iterations)
        least_squares_cg(Cui, X, Y, lambda_val)
        least_squares_cg(Ciu, Y, X, lambda_val)

    return sparse.csr_matrix(X), sparse.csr_matrix(Y)


def least_squares_cg(Cui, X, Y, lambda_val, cg_steps=3):
    users, features = X.shape

    YtY = Y.T.dot(Y) + lambda_val * np.eye(features)

    for u in xrange(users):

        x = X[u]
        r = -YtY.dot(x)

        for i, confidence in nonzeros(Cui, u):
            r += (confidence - (confidence - 1) * Y[i].dot(x)) * Y[i]

        p = r.copy()
        rsold = r.dot(r)

        for it in xrange(cg_steps):
            Ap = YtY.dot(p)
            for i, confidence in nonzeros(Cui, u):
                Ap += (confidence - 1) * Y[i].dot(p) * Y[i]

            alpha = rsold / p.dot(Ap)
            x += alpha * p
            r -= alpha * Ap

            rsnew = r.dot(r)
            p = r + (rsnew / rsold) * p
            rsold = rsnew

        X[u] = x

alpha_val = 15
conf_data = (data_sparse * alpha_val).astype('double')
user_vecs, item_vecs = implicit_als_cg(conf_data, iterations=20, features=20)

```

```
58
```

Running this for 20 iteration on the same 100K rows took only 1.5 minutes!

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*.

```
import sys
import pandas as pd
import numpy as np
import scipy.sparse as sparse
from scipy.sparse.linalg import spsolve
import random

from sklearn.preprocessing import MinMaxScaler

import implicit # The Cython library

# Load the data like we did before
raw_data = pd.read_table('data/usersha1-artmbid-artname-plays.tsv')
raw_data = raw_data.drop(raw_data.columns[1], axis=1)
raw_data.columns = ['user', 'artist', 'plays']

# Drop NaN columns
data = raw_data.dropna()
data = data.copy()

# Create a numeric user_id and artist_id column
data['user'] = data['user'].astype("category")
data['artist'] = data['artist'].astype("category")
data['user_id'] = data['user'].cat.codes
data['artist_id'] = data['artist'].cat.codes

# The implicit library expects data as a item-user matrix so we
# create two matricies, one for fitting the model (item-user)
# and one for recommendations (user-item)
sparse_item_user = sparse.csr_matrix((data['plays'].astype(float), (data['artist_id'], data['us
sparse_user_item = sparse.csr_matrix((data['plays'].astype(float), (data['user_id'], data['arti
```

```
# Initialize the als model and fit it using the sparse item-user matrix
model = implicit.als.AlternatingLeastSquares(factors=20, regularization=0.1, iterations=20)

# Calculate the confidence by multiplying it by our alpha value.
alpha_val = 15
data_conf = (sparse_item_user * alpha_val).astype('double')

# Fit the model
model.fit(data_conf)


#---------------------
# FIND SIMILAR ITEMS
#---------------------

# Find the 10 most similar to Jay-Z
item_id = 147068 #Jay-Z
n_similar = 10

# Get the user and item vectors from our trained model
user_vecs = model.user_factors
item_vecs = model.item_factors

# Calculate the vector norms
item_norms = np.sqrt((item_vecs * item_vecs).sum(axis=1))

# Calculate the similarity score, grab the top N items and
# create a list of item-score tuples of most similar artists
scores = item_vecs.dot(item_vecs[item_id]) / item_norms
top_idx = np.argpartition(scores, -n_similar)[-n_similar:]
similar = sorted(zip(top_idx, scores[top_idx] / item_norms[item_id]), key=lambda x: -x[1])

# Print the names of our most similar artists
for item in similar:
    idx, score = item
    print data.artist.loc[data.artist_id == idx].iloc[0]


#------------------------------
# CREATE USER RECOMMENDATIONS
#------------------------------

def recommend(user_id, sparse_user_item, user_vecs, item_vecs, num_items=10):
    """The same recommendation function we used before"""
```

```
    user_interactions = sparse_user_item[user_id,:].toarray()

    user_interactions = user_interactions.reshape(-1) + 1
    user_interactions[user_interactions > 1] = 0

    rec_vector = user_vecs[user_id,:].dot(item_vecs.T).toarray()

    min_max = MinMaxScaler()
    rec_vector_scaled = min_max.fit_transform(rec_vector.reshape(-1,1))[:,0]
    recommend_vector = user_interactions * rec_vector_scaled

    item_idx = np.argsort(recommend_vector)[::-1][:num_items]

    artists = []
    scores = []

    for idx in item_idx:
        artists.append(data.artist.loc[data.artist_id == idx].iloc[0])
        scores.append(recommend_vector[idx])

    recommendations = pd.DataFrame({'artist': artists, 'score': scores})

    return recommendations

# Get the trained user and item vectors. We convert them to
# csr matrices to work with our previous recommend function.
user_vecs = sparse.csr_matrix(model.user_factors)
item_vecs = sparse.csr_matrix(model.item_factors)

# Create recommendations for user with id 2025
user_id = 2025

recommendations = recommend(user_id, sparse_user_item, user_vecs, item_vecs)

print recommendations
```

Implicit also has built in functions for recommendations and similar items so we can scale it all down to just a few source lines of code:

```
import sys
```

```
import pandas as pd
import numpy as np
import scipy.sparse as sparse
from scipy.sparse.linalg import spsolve
import random

from sklearn.preprocessing import MinMaxScaler

import implicit

# Load the data like we did before
raw_data = pd.read_table('data/usersha1-artmbid-artname-plays.tsv')
raw_data = raw_data.drop(raw_data.columns[1], axis=1)
raw_data.columns = ['user', 'artist', 'plays']

# Drop NaN columns
data = raw_data.dropna()
data = data.copy()

# Create a numeric user_id and artist_id column
data['user'] = data['user'].astype("category")
data['artist'] = data['artist'].astype("category")
data['user_id'] = data['user'].cat.codes
data['artist_id'] = data['artist'].cat.codes

# The implicit library expects data as a item-user matrix so we
# create two matricies, one for fitting the model (item-user)
# and one for recommendations (user-item)
sparse_item_user = sparse.csr_matrix((data['plays'].astype(float), (data['artist_id'], data['use
sparse_user_item = sparse.csr_matrix((data['plays'].astype(float), (data['user_id'], data['artis

# Initialize the als model and fit it using the sparse item-user matrix
model = implicit.als.AlternatingLeastSquares(factors=20, regularization=0.1, iterations=20)

# Calculate the confidence by multiplying it by our alpha value.
alpha_val = 15
data_conf = (sparse_item_user * alpha_val).astype('double')

#Fit the model
model.fit(data_conf)


#---------------------
# FIND SIMILAR ITEMS
```

```
#---------------------

# Find the 10 most similar to Jay-Z
item_id = 147068 #Jay-Z
n_similar = 10

# Use implicit to get similar items.
similar = model.similar_items(item_id, n_similar)

# Print the names of our most similar artists
for item in similar:
    idx, score = item
    print data.artist.loc[data.artist_id == idx].iloc[0]


#------------------------------
# CREATE USER RECOMMENDATIONS
#------------------------------

# Create recommendations for user with id 2025
user_id = 2025

# Use the implicit recommender.
recommended = model.recommend(user_id, sparse_user_item)

artists = []
scores = []

# Get artist names from ids
for item in recommended:
    idx, score = item
    artists.append(data.artist.loc[data.artist_id == idx].iloc[0])
    scores.append(score)

# Create a dataframe of artist names and scores
recommendations = pd.DataFrame({'artist': artists, 'score': scores})

print recommendations

```

That's it! A slow and Verbose, a slightly faster and more compact and a fast and user friendly way to implement Alternating Least Squares with implicit data in Python.

References and resources:

- http://yifanhu.net/PUB/cf.pdf
- https://jessesw.com/Rec-System/
- https://github.com/benfred/implicit
- http://www.benfrederickson.com/matrix-factorization/
- http://www.benfrederickson.com/fast-implicit-matrix-factorization/
- 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

Machine Learning Data Science Statistics Recommendation System Python