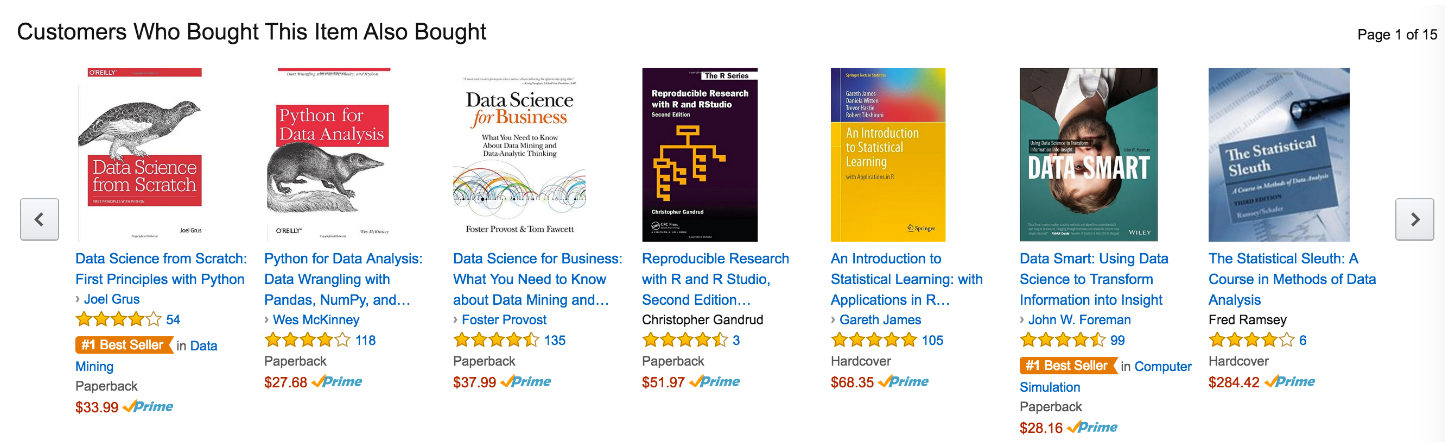
Jesse Steinweg-Woods, Ph.D.
Data Scientist

Blog About Book Reviews Websites

A Gentle Introduction to Recommender Systems with Implicit Feedback


Customers Who Bought This Item Also Bought

Page 1 of 15

Data Science from Scratch: First Principles with Python
› Joel Grus
54
#1 Best Seller in Data Mining
Paperback
$33.99 Prime

Python for Data Analysis: Data Wrangling with Pandas, NumPy, and…
› Wes McKinney
118
Paperback
$27.68 Prime

Data Science for Business: What You Need to Know about Data Mining and…
› Foster Provost
135
Paperback
$37.99 Prime

Reproducible Research with R and R Studio, Second Edition…
Christopher Gandrud
3
Paperback
$51.97 Prime

An Introduction to Statistical Learning: with Applications in R…
› Gareth James
105
Hardcover
$68.35 Prime

Data Smart: Using Data Science to Transform Information into Insight
› John W. Foreman
99
#1 Best Seller in Computer Simulation
Paperback
$28.16 Prime

The Statistical Sleuth: A Course in Methods of Data Analysis
Fred Ramsey
6
Hardcover
$284.42 Prime

Recommender systems have become a very important part of the retail, social networking, and entertainment industries. From providing advice on songs for you to try, suggesting books for you to read, or finding clothes to buy, recommender systems have greatly improved the ability of customers to make choices more easily.

Why is it so important for customers to have support in decision making? A well-cited study (over 2000 citations so far!) by Iyengar and Lepper ran an experiment where they had two stands of jam on two different days. One stand had 24 varieties of jam while the second had only six. The stand with 24 varieties of jam only converted 3% of the customers to a sale, while the stand with only six varieties converted 30% of the customers. This was an increase in sales of nearly ten-fold!

Given the number of possible choices available, especially for online shopping, having some extra guidance on these choices can really make a difference. Xavier Amatriain, now at Quora and previously at Netflix, gave an absolutely outstanding talk on recommender systems at Carnegie Mellon in 2014. I have included the talk below if you would like to see it.

Some of the key statistics about recommender systems he notes are the following:

- At Netflix, 2/3 of the movies watched are recommended
- At Google, news recommendations improved click-through rate (CTR) by 38%
- For Amazon, 35% of sales come from recommendations

In addition, at Hulu, incorporating a recommender system improved their CTR by three times over just recommending the most popular shows back in 2011. When well implemented, recommender systems can give your company a great edge.

That doesn't mean your company should necessarily build one, however. Valerie Coffman posted this article back in 2013, explaining that you need a fairly large amount of data on your customers and product purchases in order to have enough information for an effective recommender system. It's not for everyone, but if you have enough data, it's a good idea to consider it.

So, let's assume you do have enough data on your customers and items to go about building one. How would you do it? Well, that depends a lot on several factors, such as:

- The kind of data you have about your users/items
- Ability to scale
- Recommendation transparency

I will cover a few of the options available and reveal the basic methodology behind each one.

Content Based (Pandora)

In the case of Pandora, the online streaming music company, they decided to engineer features from all of the songs in their catalog as part of the Music Genome Project. Most songs are based on a feature vector of approximately 450 features, which were derived in a very long and arduous process. Once you have this feature set, one technique that works well enough is to treat the recommendation problem as a binary classification problem. This allows one to use more traditional machine learning techniques that output a probability for a certain user to like a specific song based on a training set of their song listening history. Then, simply recommend the songs with the greatest probability of being liked.

Most of the time, however, you aren't going to have features already encoded for all of your products. This would be very difficult, and it took Pandora several years to finish so it probably won't be a great option.

Demographic Based (Facebook)

If you have a lot of demographic information about your users like Facebook or LinkedIn does, you may be able to recommend based on similar users and their past behavior. Similar to the content based method, you could derive a feature vector for each of your users and generate models that predict probabilities of liking certain items.

Again, this requires a lot of information about your users that you probably don't have in most cases.

So if you need a method that doesn't care about detailed information regarding your items or your users, collaborative filtering is a very powerful method that works with surprising efficacy.

Collaborative Filtering

This is based on the relationship between users and items, with no information about the users or the items required! All you need is a rating of some kind for each user/item interaction that occurred where available. There are two kinds of data available for this type of interaction: explicit and implicit.

- Explicit: A score, such as a rating or a like
- Implicit: Not as obvious in terms of preference, such as a click, view, or purchase

The most common example discussed is movie ratings, which are given on a numeric scale. We can easily see whether a user enjoyed a movie based on the rating provided. The problem, however, is that most of the time, people don't provide ratings at all (I am totally guilty of this on Netflix!), so the amount of data available is quite scarce. Netflix at least knows whether I watched something, which requires no further input on my part. It may be the case that I watched something but didn't like it afterwards. So, it can be more difficult to infer whether this type of movie should be considered a positive recommendation or not.

Regardless of this disadvantage, implicit feedback is usually the way to go. Hulu, in a blog post about their recommendation system states:

> *As the quantity of implicit data at Hulu far outweighs the amount of explicit feedback, our system should be designed primarily to work with implicit feedback data.*

Since more data usually means a better model, implicit feedback is where our efforts should be focused. While there are a variety of ways to tackle collaborative filtering with implicit feedback, I will focus on the method included in Spark's library used for collaborative filtering, alternating least squares (ALS).

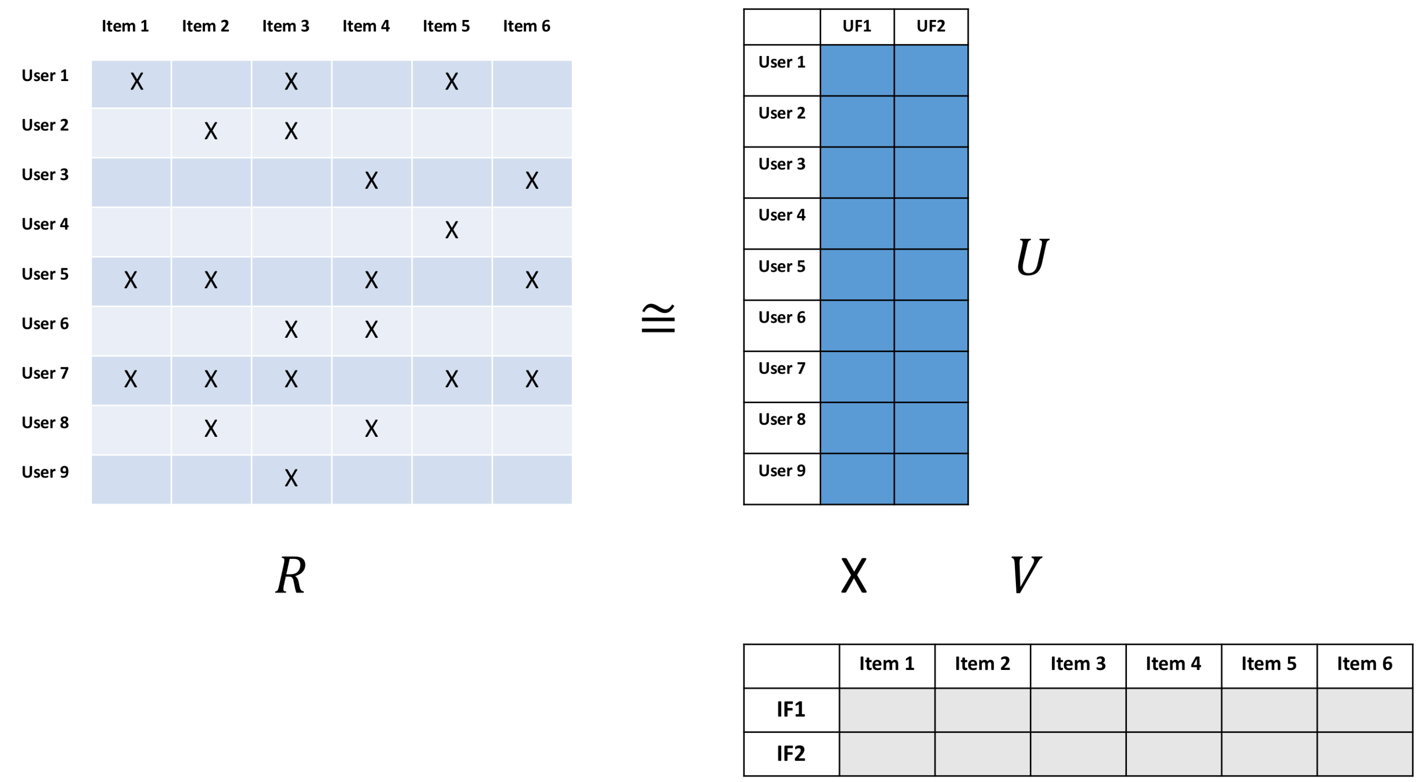
Alternating Least Squares

Before we start building our own recommender system on an example problem, I want to explain some of the intuition behind how this method works and why it likely is the only chosen method in Spark's library. We discussed before how collaborative filtering doesn't require any information about the users or items. Well, is there another way we can figure out how the users and the items are related to each other?

It turns out we can if we apply matrix factorization. Often, matrix factorization is applied in the realm of dimensionality reduction, where we are trying to reduce the number of features while still keeping the relevant information. This is the case with principal component analysis (PCA) and the very similar singular value decomposition (SVD).

Essentially, can we take a large matrix of user/item interactions and figure out the latent (or hidden) features that relate them to each other in a much smaller matrix of user features and item features? That's exactly what ALS is trying to do through matrix factorization.

As the image below demonstrates, let's assume we have an original ratings matrix R of size MxN, where M is the number of users and N is the number of items. This matrix is quite sparse, since most users only interact with a few items each. We can factorize this matrix into two separate smaller matrices: one with dimensions MxK which will be our latent user feature vectors for each user (U) and a second with dimensions KxN, which will have our latent item feature vectors for each item (V). Multiplying these two feature matrices together approximates the original matrix, but now we have two matrices that are dense including a number of latent features K for each of our items and users.



In order to solve for U and V, we could either utilize SVD (which would require inverting a potentially very large matrix and be computationally expensive) to solve the factorization more precisely or apply ALS to approximate it. In the case of ALS, we only need to solve one feature vector at a time, which means it can be run in parallel! (This large advantage is probably why it is the method of choice for Spark). To do this, we can randomly initialize U and solve for V. Then we can go back and solve for U using our solution for V. Keep iterating back and forth like this until we get a convergence that approximates R as best as we can.

After this has been finished, we can simply take the dot product of U and V to see what the predicted rating would be for a specific user/item interaction, even if there was no prior interaction. This basic methodology was adopted for implicit feedback problems in the paper Collaborative Filtering for Implicit Feedback Datasets by Hu, Koren, and Volinsky.

We will use this paper's method on a real dataset and build our own recommender system.

Processing the Data

The data we are using for this example comes from the infamous UCI Machine Learning repository. The dataset is called "Online Retail" and is found here. As you can see in the description, this dataset contains all purchases made for an online retail company based in the UK during an eight month period.

We need to take all of the transactions for each customer and put these into a format ALS can use. This means we need each unique customer ID in the rows of the matrix, and each unique item ID in the columns of the matrix. The values of the matrix should be the total number of purchases for each item by each customer.

First, let's load some libraries that will help us out with the preprocessing step. Pandas is always helpful!

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

The first step is to load the data in. Since the data is saved in an Excel file, we can use Pandas to load it.

```
website_url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/00352/Onli
retail_data = pd.read_excel(website_url) # This may take a couple minutes
```

Now that the data has been loaded, we can see what is in it.

```
retail_data.head()
```

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	2010-12-01 08:26:00	2.55	17850.0	United Kingdom
1	536365	71053	WHITE METAL LANTERN	6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	2010-12-01 08:26:00	2.75	17850.0	United Kingdom
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom

The dataset includes the invoice number for different purchases, along with the StockCode (or item ID), an item description, the number purchased, the date of purchase, the price of the items, a customer ID, and the country of origin for the customer.

Let’s check to see if there are any missing values in the data.

```
retail_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 541909 entries, 0 to 541908
Data columns (total 8 columns):
InvoiceNo      541909 non-null object
StockCode      541909 non-null object
Description    540455 non-null object
Quantity       541909 non-null int64
InvoiceDate    541909 non-null datetime64[ns]
UnitPrice      541909 non-null float64
```

```
CustomerID     406829 non-null float64
Country        541909 non-null object
dtypes: datetime64[ns](1), float64(2), int64(1), object(4)
memory usage: 33.1+ MB
```

Most columns have no missing values, but Customer ID is missing in several rows. If the customer ID is missing, we don't know who bought the item. We should drop these rows from our data first. We can use the pd.isnull to test for rows with missing data and only keep the rows that have a customer ID.

```
cleaned_retail = retail_data.loc[pd.isnull(retail_data.CustomerID) == False]
```

```
cleaned_retail.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 406829 entries, 0 to 541908
Data columns (total 8 columns):
InvoiceNo      406829 non-null object
StockCode      406829 non-null object
Description    406829 non-null object
Quantity       406829 non-null int64
InvoiceDate    406829 non-null datetime64[ns]
UnitPrice      406829 non-null float64
CustomerID     406829 non-null float64
Country        406829 non-null object
dtypes: datetime64[ns](1), float64(2), int64(1), object(4)
memory usage: 27.9+ MB
```

Much better. Now we have no missing values and all of the purchases can be matched to a specific customer.

Before we make any sort of ratings matrix, it would be nice to have a lookup table that keeps track of each item ID along with a description of that item. Let's make that now.

```
item_lookup = cleaned_retail[['StockCode', 'Description']].drop_duplicates() # Onl
item_lookup['StockCode'] = item_lookup.StockCode.astype(str) # Encode as strings f
```

```
item_lookup.head()
```

StockCode	Description
085123A	WHITE HANGING HEART T-LIGHT HOLDER
171053	WHITE METAL LANTERN
284406B	CREAM CUPID HEARTS COAT HANGER
384029G	KNITTED UNION FLAG HOT WATER BOTTLE
484029E	RED WOOLLY HOTTIE WHITE HEART.

This can tell us what each item is, such as that StockCode 71053 is a white metal lantern. Now that this has been created, we need to:

- Group purchase quantities together by stock code and item ID
- Change any sums that equal zero to one (this can happen if items were returned, but we want to indicate that the user actually purchased the item instead of assuming no interaction between the user and the item ever took place)
- Only include customers with a positive purchase total to eliminate possible errors
- Set up our sparse ratings matrix

This last step is especially important if you don't want to have unnecessary memory issues! If you think about it, our matrix is going to contain thousands of items and thousands of users with a user/item value required for every possible combination. That is a LARGE matrix, so we can save a lot of memory by keeping the matrix sparse and only saving the locations and values of items that are not zero.

The code below will finish the preprocessing steps necessary for our final ratings sparse matrix:

```
cleaned_retail['CustomerID'] = cleaned_retail.CustomerID.astype(int) # Convert to
cleaned_retail = cleaned_retail[['StockCode', 'Quantity', 'CustomerID']] # Get rid
grouped_cleaned = cleaned_retail.groupby(['CustomerID', 'StockCode']).sum().reset_
grouped_cleaned.Quantity.loc[grouped_cleaned.Quantity == 0] = 1 # Replace a sum of
# indicate purchased
grouped_purchased = grouped_cleaned.query('Quantity > 0') # Only get customers whe
```

If we look at our final resulting matrix of grouped purchases, we see the following:

```
grouped_purchased.head()
```

CustomerID	StockCode	Quantity
012346	23166	1
112347	16008	24
212347	17021	36
312347	20665	6
412347	20719	40

Instead of representing an explicit rating, the purchase quantity can represent a "confidence" in terms of how strong the interaction was. Items with a larger number of purchases by a customer can carry more weight in our ratings matrix of purchases.

Our last step is to create the sparse ratings matrix of users and items utilizing the code below:

```
customers = list(np.sort(grouped_purchased.CustomerID.unique())) # Get our unique
products = list(grouped_purchased.StockCode.unique()) # Get our unique products th
quantity = list(grouped_purchased.Quantity) # All of our purchases

rows = grouped_purchased.CustomerID.astype('category', categories = customers).cat
# Get the associated row indices
cols = grouped_purchased.StockCode.astype('category', categories = products).cat.c
```

```
# Get the associated column indices
purchases_sparse = sparse.csr_matrix((quantity, (rows, cols)), shape=(len(customer
```

Let’s check our final matrix object:

```
purchases_sparse
```

```
<4338x3664 sparse matrix of type '<class 'numpy.int64'>'
	with 266723 stored elements in Compressed Sparse Row format>
```

We have 4338 customers with 3664 items. For these user/item interactions, 266723 of these items had a purchase. In terms of sparsity of the matrix, that makes:

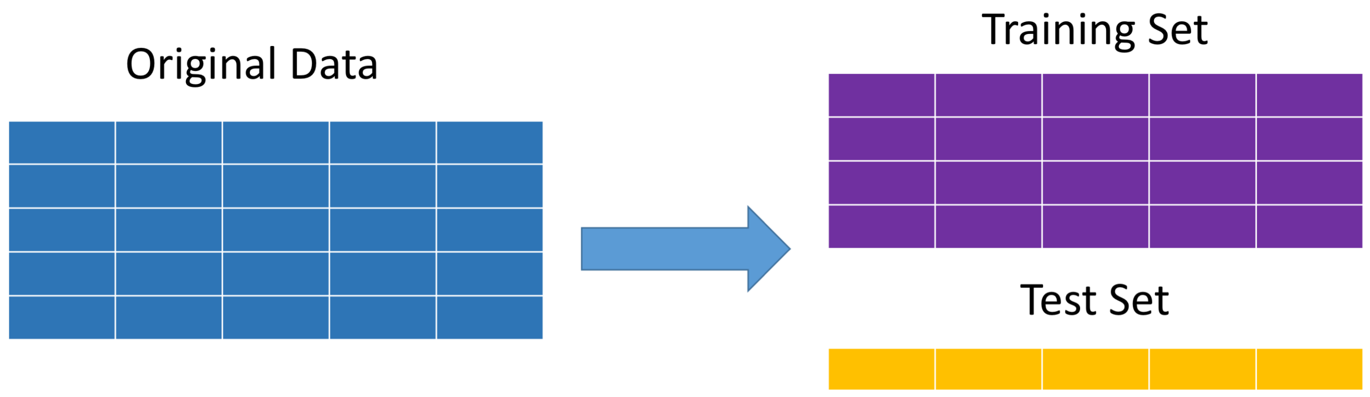
```
matrix_size = purchases_sparse.shape[0]*purchases_sparse.shape[1] # Number of poss
num_purchases = len(purchases_sparse.nonzero()[0]) # Number of items interacted wi
sparsity = 100*(1 - (num_purchases/matrix_size))
sparsity
```

```
98.32190920694744
```

98.3% of the interaction matrix is sparse. For collaborative filtering to work, the maximum sparsity you could get away with would probably be about 99.5% or so. We are well below this, so we should be able to get decent results.

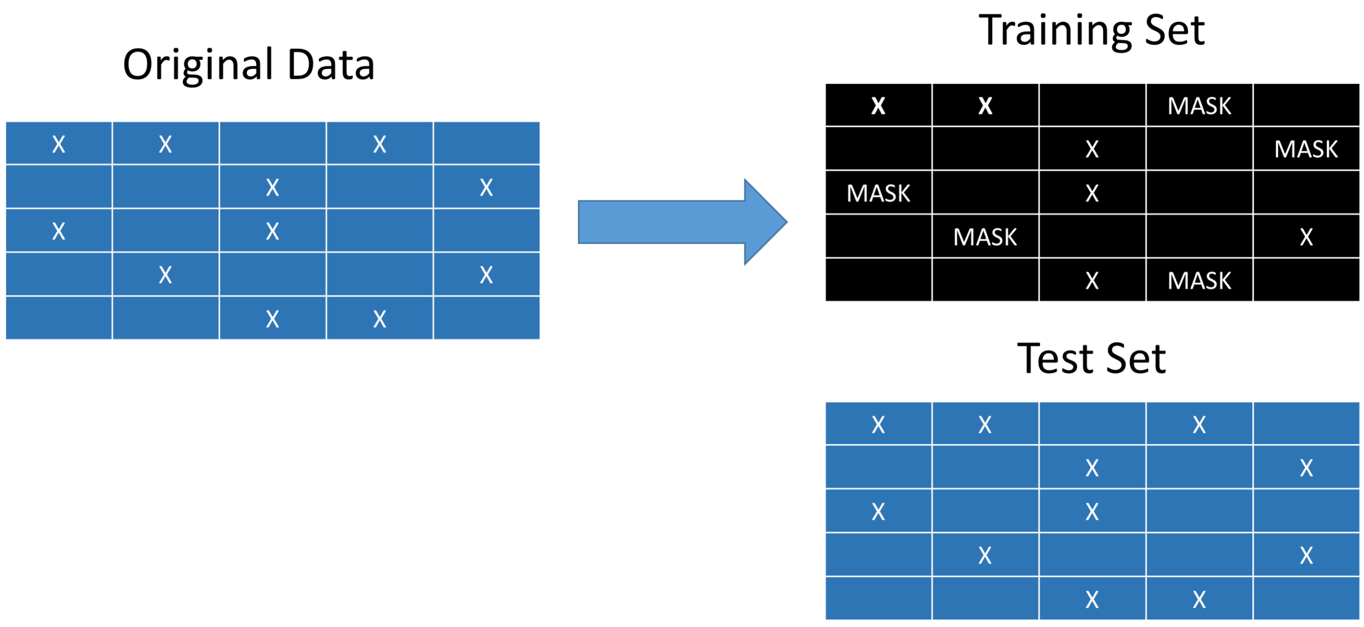
Creating a Training and Validation Set

Typically in Machine Learning applications, we need to test whether the model we just trained is any good on new data it hasn't yet seen before from the training phase. We do this by creating a test set completely separate from the training set. Usually this is fairly simple: just take a random sample of the training example rows in our feature matrix and separate it away from the training set. That normally looks like this:



With collaborative filtering, that's not going to work because you need all of the user/item interactions to find the proper matrix factorization. A better method is to hide a certain percentage of the user/item interactions from the model during the training phase chosen at random. Then, check during the test phase how many of the items that were recommended the user actually ended up purchasing in the end. Ideally, you would ultimately test your recommendations with some kind of A/B test or utilizing data from a time series where all data prior to a certain point in time is used for training while data after a certain period of time is used for testing.

For this example, because the time period is only 8 months and because of the purchasing type (products), it is most likely products won't be purchased again in a short time period anyway. This will be a better test. You can see an example here:



Our test set is an exact copy of our original data. The training set, however, will mask a random percentage of user/item interactions and act as if the user never purchased the item (making it a sparse entry with a zero). We then check in the test set which items were recommended to the user that they ended up actually purchasing. If the users frequently ended up purchasing the items most recommended to them by the system, we can conclude the system seems to be working.

As an additional check, we can compare our system to simply recommending the most popular items to every user (beating popularity is a bit difficult). This will be our baseline.

This method of testing isn't necessarily the “correct” answer, because it depends on how you want to use the recommender system. However, it is a practical way of testing performance I will use for this example.

Now that we have a plan on how to separate our training and testing sets, let's create a function that can do this for us. We will also import the random library and set a seed so that you will see the same results as I did.

```python
import random
```

```
def make_train(ratings, pct_test = 0.2):
    '''
    This function will take in the original user-item matrix and "mask" a percenta
    user-item interaction has taken place for use as a test set. The test set will
    while the training set replaces the specified percentage of them with a zero i

    parameters:

    ratings - the original ratings matrix from which you want to generate a train/
    copy of the original set. This is in the form of a sparse csr_matrix.

    pct_test - The percentage of user-item interactions where an interaction took
    training set for later comparison to the test set, which contains all of the o

    returns:

    training_set - The altered version of the original data with a certain percent
    that originally had interaction set back to zero.

    test_set - A copy of the original ratings matrix, unaltered, so it can be used
    compares with the actual interactions.

    user_inds - From the randomly selected user-item indices, which user rows were
    This will be necessary later when evaluating the performance via AUC.
    '''
    test_set = ratings.copy() # Make a copy of the original set to be the test set
    test_set[test_set != 0] = 1 # Store the test set as a binary preference matrix
    training_set = ratings.copy() # Make a copy of the original data we can alter
    nonzero_inds = training_set.nonzero() # Find the indices in the ratings data w
    nonzero_pairs = list(zip(nonzero_inds[0], nonzero_inds[1])) # Zip these pairs
    random.seed(0) # Set the random seed to zero for reproducibility
    num_samples = int(np.ceil(pct_test*len(nonzero_pairs))) # Round the number of
    samples = random.sample(nonzero_pairs, num_samples) # Sample a random number o
    user_inds = [index[0] for index in samples] # Get the user row indices
    item_inds = [index[1] for index in samples] # Get the item column indices
    training_set[user_inds, item_inds] = 0 # Assign all of the randomly chosen use
    training_set.eliminate_zeros() # Get rid of zeros in sparse array storage afte
    return training_set, test_set, list(set(user_inds)) # Output the unique list o
```

This will return our training set, a test set that has been binarized to 0/1 for purchased/not purchased, and a list of which users had at least one item masked. We will test the performance of the recommender system on these users only. I am masking 20% of the user/item interactions for this example.

```
product_train, product_test, product_users_altered = make_train(purchases_sparse,
```

Now that we have our train/test split, it is time to implement the alternating least squares algorithm from the Hu, Koren, and Volinsky paper.

Implementing ALS for Implicit Feedback

Now that we have our training and test sets finished, we can move on to implementing the algorithm. If you look at the paper previously linked above

- Hu, Koren, and Volinsky

you can see the key equations will we need to implement into the algorithm. First, we have our ratings matrix which is sparse (represented by the product_train sparse matrix object). We need to turn this into a confidence matrix (from page 4):

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

Where C_{ui} is the confidence matrix for our users u and our items i. The α term represents a linear scaling of the rating preferences (in our case number of purchases) and the r_{ui} term is our original matrix of purchases. The paper suggests 40 as a good starting point.

After taking the derivative of equation 3 in the paper, we can minimize the cost function for our users U:

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

The authors note you can speed up this computation through some linear algebra that changes this equation to:

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

Notice that we can now precompute the Y^TY portion without having to iterate through each user u. We can derive a similar equation for our items:

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

These will be the two equations we will iterate back and forth between until they converge. We also have a regularization term λ that can help prevent overfitting during the training stage as well, along with our binarized preference matrix p which is just 1 where there was a purchase (or interaction) and zero where there was not.

Now that the math part is out of the way, we can turn this into code! Shoutout to Chris Johnson's implicit-mf code that was a helpful guide for this. I have altered his to make things easier to understand.

```
def implicit_weighted_ALS(training_set, lambda_val = 0.1, alpha = 40, iterations =
    '''
    Implicit weighted ALS taken from Hu, Koren, and Volinsky 2008. Designed for al
    feedback based collaborative filtering.

    parameters:

    training_set - Our matrix of ratings with shape m x n, where m is the number o
    Should be a sparse csr matrix to save space.

    lambda_val - Used for regularization during alternating least squares. Increas
    but decrease variance. Default is 0.1.

    alpha - The parameter associated with the confidence matrix discussed in the p
    The paper found a default of 40 most effective. Decreasing this will decrease
    various ratings.

    iterations - The number of times to alternate between both user feature vector
    alternating least squares. More iterations will allow better convergence at th
    The authors found 10 iterations was sufficient, but more may be required to co

    rank_size - The number of latent features in the user/item feature vectors. Th
    between 20-200. Increasing the number of features may overfit but could reduce
```

```
seed - Set the seed for reproducible results

returns:

The feature vectors for users and items. The dot product of these feature vect
"rating" at each point in your original matrix.
'''

# first set up our confidence matrix

conf = (alpha*training_set) # To allow the matrix to stay sparse, I will add o
                            # and converted to dense.
num_user = conf.shape[0]
num_item = conf.shape[1] # Get the size of our original ratings matrix, m x n

# initialize our X/Y feature vectors randomly with a set seed
rstate = np.random.RandomState(seed)

X = sparse.csr_matrix(rstate.normal(size = (num_user, rank_size))) # Random nu
Y = sparse.csr_matrix(rstate.normal(size = (num_item, rank_size))) # Normally
                                                                 # transpose at th
X_eye = sparse.eye(num_user)
Y_eye = sparse.eye(num_item)
lambda_eye = lambda_val * sparse.eye(rank_size) # Our regularization term lamb

# We can compute this before iteration starts.

# Begin iterations

for iter_step in range(iterations): # Iterate back and forth between solving X
    # Compute yTy and xTx at beginning of each iteration to save computing tim
    yTy = Y.T.dot(Y)
    xTx = X.T.dot(X)
    # Being iteration to solve for X based on fixed Y
    for u in range(num_user):
        conf_samp = conf[u,:].toarray() # Grab user row from confidence matrix
        pref = conf_samp.copy()
        pref[pref != 0] = 1 # Create binarized preference vector
        CuI = sparse.diags(conf_samp, [0]) # Get Cu - I term, don't need to su
```

```
            yTCuIY = Y.T.dot(CuI).dot(Y) # This is the yT(Cu-I)Y term
            yTCupu = Y.T.dot(CuI + Y_eye).dot(pref.T) # This is the yTCuPu term, w
                                                      # Cu - I + I = Cu
            X[u] = spsolve(yTy + yTCuIY + lambda_eye, yTCupu)
            # Solve for Xu = ((yTy + yT(Cu-I)Y + lambda*I)^-1)yTCuPu, equation 4 f
        # Begin iteration to solve for Y based on fixed X
        for i in range(num_item):
            conf_samp = conf[:,i].T.toarray() # transpose to get it in row format
            pref = conf_samp.copy()
            pref[pref != 0] = 1 # Create binarized preference vector
            CiI = sparse.diags(conf_samp, [0]) # Get Ci - I term, don't need to su
            xTCiIX = X.T.dot(CiI).dot(X) # This is the xT(Cu-I)X term
            xTCiPi = X.T.dot(CiI + X_eye).dot(pref.T) # This is the xTCiPi term
            Y[i] = spsolve(xTx + xTCiIX + lambda_eye, xTCiPi)
            # Solve for Yi = ((xTx + xT(Cu-I)X) + lambda*I)^-1)xTCiPi, equation 5
    # End iterations
    return X, Y.T # Transpose at the end to make up for not being transposed at th
                          # Y needs to be rank x n. Keep these as separate matrices
```

Hopefully the comments are enough to see how the code was structured. You want to keep the matrices sparse where possible to avoid memory issues! Let's try just a single iteration of the code to see how it works (it's pretty slow right now!) I will choose 20 latent factors as my rank matrix size along with an alpha of 15 and regularization of 0.1 (which I found in testing does the best). This takes about 90 seconds to run on my MacBook Pro.

```
user_vecs, item_vecs = implicit_weighted_ALS(product_train, lambda_val = 0.1, alph
                                             rank_size = 20)
```

We can investigate ratings for a particular user by taking the dot product between the user and item vectors (U and V). Let's look at our first user.

```
user_vecs[0,:].dot(item_vecs).toarray()[0,:5]
```

```
array([ 0.00644811, -0.0014369 ,  0.00494281,  0.00027502,  0.01275582])
```

This is a sample of the first five items out of the 3664 in our stock. The first user in our matrix has the fifth item with the greatest recommendation out of the first five items. However, notice we only did one iteration because our algorithm was so slow! You should iterate at least ten times according to the authors so that U and V converge. We could wait 15 minutes to let this run, or . . . use someone else's code that is much faster!

Speeding Up ALS

This code in its raw form is just too slow. We have to do a lot of looping, and we haven't taken advantage of the fact that our algorithm is embarrasingly parallel, since we could do each iteration of the item and user vectors independently. Fortunately, as I was still finishing this up, Ben Frederickson at Flipboard had perfect timing and came out with a version of ALS for Python utilizing Cython and parallelizing the code among threads. You can read his blog post about using it for finding similar music artists using matrix factorization here and his implicit library here. He claims it is even faster than Quora's C++ QMF, but I haven't tried theirs. All I can tell you is that it is over 1000 times faster than this bare bones pure Python version when I tested it. Install this library before you continue and follow the instructions. If you have conda installed, just do pip install implicit and you should be good to go.

First, import his library so we can utilize it for our matrix factorization.

```
import implicit
```

His version of the code doesn't have a parameter for the weighting α and assumes you are doing this to the ratings matrix before using it as an input. I did some testing and found the following settings to work the best. Also make sure that we set the type of our matrix to double for the ALS function to run properly.

```
alpha = 15
user_vecs, item_vecs = implicit.alternating_least_squares((product_train*alpha).as
                                                      factors=20,
                                                      regularization = 0.1,
                                                     iterations = 50)
```

Much faster, right? We now have recommendations for all of our users and items. However, how do we know if these are any good?

Evaluating the Recommender System

Remember that our training set had 20% of the purchases masked? This will allow us to evaluate the performance of our recommender system. Essentially, we need to see if the order of recommendations given for each user matches the items they ended up purchasing. A commonly used metric for this kind of problem is the area under the Receiver Operating Characteristic (or ROC) curve. A greater area under the curve means we are recommending items that end up being purchased near the top of the list of recommended items. Usually this metric is used in more typical binary classification problems to identify how well a model can predict a positive example vs. a negative one. It will also work well for our purposes of ranking recommendations.

In order to do that, we need to write a function that can calculate a mean area under the curve (AUC) for any user that had at least one masked item. As a benchmark, we will also calculate what the mean AUC would have been if we had simply recommended the most popular items. Popularity tends to be hard to beat in most recommender system problems, so it makes a good comparison.

First, let's make a simple function that can calculate our AUC. Scikit-learn has one we can alter a bit.

```
from sklearn import metrics
```

```
def auc_score(predictions, test):
    '''
    This simple function will output the area under the curve using sklearn's metr
    parameters:

    - predictions: your prediction output

    - test: the actual target result you are comparing to

    returns:

    - AUC (area under the Receiver Operating Characterisic curve)
    '''
    fpr, tpr, thresholds = metrics.roc_curve(test, predictions)
    return metrics.auc(fpr, tpr)
```

Now, utilize this helper function inside of a second function that will calculate the AUC for each user in our training set that had at least one item masked. It should also calculate AUC for the most popular items for our users to compare.

```
def calc_mean_auc(training_set, altered_users, predictions, test_set):
    '''
    This function will calculate the mean AUC by user for any user that had their

    parameters:

    training_set - The training set resulting from make_train, where a certain per
    user/item interactions are reset to zero to hide them from the model

    predictions - The matrix of your predicted ratings for each user/item pair as
    These should be stored in a list, with user vectors as item zero and item vect

    altered_users - The indices of the users where at least one user/item pair was

    test_set - The test set constucted earlier from make_train function



    returns:
```

```
    The mean AUC (area under the Receiver Operator Characteristic curve) of the te
    there were originally zero to test ranking ability in addition to the most pop
    '''



    store_auc = [] # An empty list to store the AUC for each user that had an item
    popularity_auc = [] # To store popular AUC scores
    pop_items = np.array(test_set.sum(axis = 0)).reshape(-1) # Get sum of item ite
    item_vecs = predictions[1]
    for user in altered_users: # Iterate through each user that had an item altere
        training_row = training_set[user,:].toarray().reshape(-1) # Get the traini
        zero_inds = np.where(training_row == 0) # Find where the interaction had n
        # Get the predicted values based on our user/item vectors
        user_vec = predictions[0][user,:]
        pred = user_vec.dot(item_vecs).toarray()[0,zero_inds].reshape(-1)
        # Get only the items that were originally zero
        # Select all ratings from the MF prediction for this user that originally
        actual = test_set[user,:].toarray()[0,zero_inds].reshape(-1)
        # Select the binarized yes/no interaction pairs from the original full dat
        # that align with the same pairs in training
        pop = pop_items[zero_inds] # Get the item popularity for our chosen items
        store_auc.append(auc_score(pred, actual)) # Calculate AUC for the given us
        popularity_auc.append(auc_score(pop, actual)) # Calculate AUC using most p
    # End users iteration

    return float('%.3f'%np.mean(store_auc)), float('%.3f'%np.mean(popularity_auc))
   # Return the mean AUC rounded to three decimal places for both test and popular
```

We can now use this function to see how our recommender system is doing. To use this function, we will need to transform our output from the ALS function to csr_matrix format and transpose the item vectors. The original pure Python version output the user and item vectors into the correct format already.

```
calc_mean_auc(product_train, product_users_altered,
              [sparse.csr_matrix(user_vecs), sparse.csr_matrix(item_vecs.T)], prod
```

```
# AUC for our recommender system
```

```
(0.87, 0.814)
```

We can see that our recommender system beat popularity. Our system had a mean AUC of 0.87, while the popular item benchmark had a lower AUC of 0.814. You can go back and tune the hyperparameters if you wish to see if you can get a higher AUC score. Ideally, you would have separate train, cross-validation, and test sets so that you aren't overfitting while tuning the hyperparameters, but this setup is adequate to demonstrate how to check that the system is working.

A Recommendation Example

We now have our recommender system trained and have proven it beats the benchmark of popularity. An AUC of 0.87 means the system is recommending items the user in fact had purchased in the test set far more frequently than items the user never ended up purchasing. To see an example of how it works, let's examine the recommendations given to a particular user and decide subjectively if they make any sense.

First, however, we need to find a way of retrieving the items already purchased by a user in the training set. Initially, we will create an array of our customers and items we made earlier.

```
customers_arr = np.array(customers) # Array of customer IDs from the ratings matri
products_arr = np.array(products) # Array of product IDs from the ratings matrix
```

Now, we can create a function that will return a list of the item descriptions from our earlier created item lookup table.

```
def get_items_purchased(customer_id, mf_train, customers_list, products_list, item
    '''
    This just tells me which items have been already purchased by a specific user

    parameters:

    customer_id - Input the customer's id number that you want to see prior purcha

    mf_train - The initial ratings training set used (without weights applied)

    customers_list - The array of customers used in the ratings matrix

    products_list - The array of products used in the ratings matrix

    item_lookup - A simple pandas dataframe of the unique product ID/product descr

    returns:

    A list of item IDs and item descriptions for a particular customer that were a
    '''
    cust_ind = np.where(customers_list == customer_id)[0][0] # Returns the index r
    purchased_ind = mf_train[cust_ind,:].nonzero()[1] # Get column indices of purc
    prod_codes = products_list[purchased_ind] # Get the stock codes for our purcha
    return item_lookup.loc[item_lookup.StockCode.isin(prod_codes)]
```

We need to look these up by a customer’s ID. Looking at the list of customers:

```
customers_arr[:5]
```

```
array([12346, 12347, 12348, 12349, 12350])
```

we can see that the first customer listed has an ID of 12346. Let’s examine their purchases from the training set.

```
get_items_purchased(12346, product_train, customers_arr, products_arr, item_lookup
```

	StockCode	Description
61619	23166	MEDIUM CERAMIC TOP STORAGE JAR

We can see that the customer purchased a ceramic jar for storage, medium size. What items does the recommender system say this customer should purchase? We need to create another function that does this. Let's also import the MinMaxScaler from scikit-learn to help with this.

```
from sklearn.preprocessing import MinMaxScaler
```

```
def rec_items(customer_id, mf_train, user_vecs, item_vecs, customer_list, item_lis
    '''
    This function will return the top recommended items to our users

    parameters:

    customer_id - Input the customer's id number that you want to get recommendati

    mf_train - The training matrix you used for matrix factorization fitting

    user_vecs - the user vectors from your fitted matrix factorization

    item_vecs - the item vectors from your fitted matrix factorization

    customer_list - an array of the customer's ID numbers that make up the rows of
                    (in order of matrix)

    item_list - an array of the products that make up the columns of your ratings
                    (in order of matrix)

    item_lookup - A simple pandas dataframe of the unique product ID/product descr
    num_items - The number of items you want to recommend in order of best recomme

    returns:

    - The top n recommendations chosen based on the user/item vectors for items ne
    '''

    cust_ind = np.where(customer_list == customer_id)[0][0] # Returns the index ro
    pref_vec = mf_train[cust_ind,:].toarray() # Get the ratings from the training 
    pref_vec = pref_vec.reshape(-1) + 1 # Add 1 to everything, so that items not p
    pref_vec[pref_vec > 1] = 0 # Make everything already purchased zero
    rec_vector = user_vecs[cust_ind,:].dot(item_vecs.T) # Get dot product of user 
    # Scale this recommendation vector between 0 and 1
    min_max = MinMaxScaler()
    rec_vector_scaled = min_max.fit_transform(rec_vector.reshape(-1,1))[:,0]
    recommend_vector = pref_vec*rec_vector_scaled
    # Items already purchased have their recommendation multiplied by zero
    product_idx = np.argsort(recommend_vector)[::-1][:num_items] # Sort the indice
    # of best recommendations
    rec_list = [] # start empty list to store items
    for index in product_idx:
        code = item_list[index]
        rec_list.append([code, item_lookup.Description.loc[item_lookup.StockCode =
        # Append our descriptions to the list
    codes = [item[0] for item in rec_list]
    descriptions = [item[1] for item in rec_list]
    final_frame = pd.DataFrame({'StockCode': codes, 'Description': descriptions})
    return final_frame[['StockCode', 'Description']] # Switch order of columns aro
```

Essentially, this will retrieve the N highest ranking dot products between our user and item vectors for a particular user. Items already purchased are not recommended to the user. For now, let's use a default of 10 items and see what the recommender system decides to pick for our customer.

```
rec_items(12346, product_train, user_vecs, item_vecs, customers_arr, products_arr,
                          num_items = 10)
```

StockCode	Description
023167	SMALL CERAMIC TOP STORAGE JAR
123165	LARGE CERAMIC TOP STORAGE JAR
222963	JAM JAR WITH GREEN LID
323294	SET OF 6 SNACK LOAF BAKING CASES
422980	PANTRY SCRUBBING BRUSH
523296	SET OF 6 TEA TIME BAKING CASES
623293	SET OF 12 FAIRY CAKE BAKING CASES
722978	PANTRY ROLLING PIN
823295	SET OF 12 MINI LOAF BAKING CASES
922962	JAM JAR WITH PINK LID

These recommendations seem quite good! Remember that the recommendation system has no real understanding of what a ceramic jar is. All it knows is the purchase history. It identified that people purchasing a medium sized jar may also want to purchase jars of a differing size. The recommender system also suggests jar magnets and a sugar dispenser, which is similar in use to a storage jar. I personally was blown away by how well the system seems to pick up on these sorts of shopping patterns. Let’s try another user that hasn’t made a large number of purchases.

```
get_items_purchased(12353, product_train, customers_arr, products_arr, item_lookup
```

StockCode	Description
214837446	MINI CAKE STAND WITH HANGING CAKES
214937449	CERAMIC CAKE STAND + HANGING CAKES
485937450	CERAMIC CAKE BOWL + HANGING CAKES
510822890	NOVELTY BISCUITS CAKE STAND 3 TIER

This person seems like they want to make cakes. What kind of items does the recommender system think they would be interested in?

```
rec_items(12353, product_train, user_vecs, item_vecs, customers_arr, products_arr,
                    num_items = 10)
```

	StockCode	Description
0	22645	CERAMIC HEART FAIRY CAKE MONEY BANK
1	22055	MINI CAKE STAND HANGING STRAWBERY
2	22644	CERAMIC CHERRY CAKE MONEY BANK
3	37447	CERAMIC CAKE DESIGN SPOTTED PLATE
4	37448	CERAMIC CAKE DESIGN SPOTTED MUG
5	22059	CERAMIC STRAWBERRY DESIGN MUG
6	22063	CERAMIC BOWL WITH STRAWBERRY DESIGN
7	22649	STRAWBERRY FAIRY CAKE TEAPOT
8	22057	CERAMIC PLATE STRAWBERRY DESIGN
9	22646	CERAMIC STRAWBERRY CAKE MONEY BANK

It certainly picked up on the cake theme along with ceramic items. Again, these recommendations seem very impressive given the system doesn't understand the content behind the recommendations. Let's try one more.

```
get_items_purchased(12361, product_train, customers_arr, products_arr, item_lookup
```

	StockCode	Description
34	22326	ROUND SNACK BOXES SET OF4 WOODLAND
35	22629	SPACEBOY LUNCH BOX
37	22631	CIRCUS PARADE LUNCH BOX
93	20725	LUNCH BAG RED RETROSPOT
369	22382	LUNCH BAG SPACEBOY DESIGN
547	22328	ROUND SNACK BOXES SET OF 4 FRUITS
549	22630	DOLLY GIRL LUNCH BOX
1241	22555	PLASTERS IN TIN STRONGMAN
58132	20725	LUNCH BAG RED SPOTTY

This customer seems like they are buying products suitable for lunch time. What other items does the recommender system think they might like?

```
rec_items(12361, product_train, user_vecs, item_vecs, customers_arr, products_arr,
                          num_items = 10)
```

```
  StockCode                          Description
0 22662      LUNCH BAG DOLLY GIRL DESIGN
1 20726      LUNCH BAG WOODLAND
2 20719      WOODLAND CHARLOTTE BAG
3 22383      LUNCH BAG SUKI DESIGN
4 20728      LUNCH BAG CARS BLUE
5 23209      LUNCH BAG DOILEY PATTERN
6 22661      CHARLOTTE BAG DOLLY GIRL DESIGN
7 20724      RED RETROSPOT CHARLOTTE BAG
8 23206      LUNCH BAG APPLE DESIGN
9 22384      LUNCH BAG PINK POLKADOT
```

Once again, the recommender system comes through! Definitely a lot of bags and lunch related items in this recommendation list. Feel free to play around with the recommendations for other users and see what the system came up with!

Summary

In this post, we have learned about how to design a recommender system with implicit feedback and how to provide recommendations. We also covered how to test the recommender system.

In real life, if the size of your ratings matrix will not fit on a single machine very easily, utilizing the implementation in Spark is going to be more practical. If you are interested in taking recommender systems to the next level, a hybrid system would be best that incorporates information about your users/items along with the purchase history. A Python library called LightFM from Maciej Kula at Lyst looks very interesting for this sort of application. You can find it here.

Last, there are several other advanced methods you can incorporate in recommender systems to get a bump in performance. Part 2 of Xavier Amatriain's lecture would be a great place to start.

If you are more interested in recommender systems with explicit feedback (such as with movie reviews) there are a couple of great posts that cover this in detail:

- Alternating Least Squares Method for Collaborative Filtering by Bugra Akyildiz
- Explicit Matrix Factorization: ALS, SGD, and All That Jazz by Ethan Rosenthal

If you are looking for great datasets to try a recommendation system out on for yourself, I found this gist helpful. Some of the links don't work anymore but it's a great place to start looking for data to try a system out on your own!

If you would like the Jupyter Notebook for this blog post, you can find it here.

Written on May 30, 2016

106 Comments **Jesse's website** 1 **Login**

Recommend 15 Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Kenny • a month ago

Hi, thanks for the great article.

One question. How would we deal with situations where we have more than one features, such as number of clicks to an item's page AND length of stay on the page? The problem arises when creating the sparse matrix, where in the example just assumed one feature, the quantity.

Thanks!

• Reply • Share ›

Jesse Steinweg-Woods Mod Kenny • a month ago

Hi Kenny,

There are a variety of ways you could try tackling this:

- Create a user/item ratings matrix with a combined click/page length "score" and factorize that
- Factorize both matrices separately (one for clicks and the other for length of stay) and try combining the predictions with a separate model to weight the recommendations accordingly between clicks/length of stay
- Just use the latent item/user vectors as features and combine all of them together in a single model. The downside to this is you will be required to calculate a score for all potential items to figure out which ones have the highest probability of being clicked.

You can decide which method works best for you. In the recommender system I built, I just treated it as a multi-class classification problem instead (no matrix factorization utilized).

1 • Reply • Share ›

melanomma ➦ Jesse Steinweg-Woods • 8 days ago

Hello Jesse! Thank you for such a great article.

I have an issue similar to Kenny's, where I have user purchases and also some user activity (clicks and others), where my goal is to recommend products, but also to interpret the activity as related to those product sales. I'm fairly new to the area and reading everything I can so I can plan how to approach this.

Could you point me to any resource documenting a case similar to what you described? (If not that's okay, thanks again for this article!!)

^ | v • Reply • Share ›

Jesse Steinweg-Woods Mod ➦ melanomma • 8 days ago

Ah. Now that's a tricky one!

If you ALSO need to interpret the activity as related to those product sales, that means you need some kind of explanation for why users made certain purchases (if I understand your meaning correctly).

One thing you could do is try to cluster the latent item vectors into a certain number of clusters (using k-means, dbscan, whatever method of clustering you prefer). You can then investigate the clusters and try to interpret what they mean (you may not be able to, in which case you will need to change the number of clusters desired).

When a user purchases a product from one of these clusters, you can use that as a categorical feature in another supervised learning model that predicts the purchases and try to interpret the coefficients from that model.

Hopefully this makes some sense to you.

Good luck!

^ | v • Reply • Share ›

melanomma ➦ Jesse Steinweg-Woods • 8 days ago

Thanks! And also for the quick reply!
So, build clusters with the item latent features, use them as categories and build a model with user activity and the category as features.

I will work on this, see where it leads :) thanks!

^ | v • Reply • Share ›

王逸峰 • a month ago

Hi Jesse,

Thank you so much for your wonderful article!

I have a question on the calc_mean_auc function, I am wondering what the pop is used for. In your code, you said pop = pop_items[zero_inds] # Get the item popularity for our chosen items

When I read your code I thought that pop is the list of items expect those items bought by our user, so do the pop and the AUC of pop be set to calculate the attraction of the rest items?

Thank you in advance!

• Reply • Share ›

Jesse Steinweg-Woods Mod 王逸峰 • a month ago

Hello,

The pop object is just the popularity of each item the user hasn't yet purchased. It's a way of recommending the most popular items to the user instead of the items determined by our dot product of the user/item latent vectors. Popularity is a strong baseline we can use as a comparison to see if our recommendations are any good.

• Reply • Share ›

王逸峰 Jesse Steinweg-Woods • a month ago

I see... Thank you for your kind reply! May I ask another question? What does the parameter alpha mean? alpha is in the alternating least square part.

Many thanks.

• Reply • Share ›

Jesse Steinweg-Woods Mod 王逸峰 • a month ago

"The α term represents a linear scaling of the rating preferences (in our case number of purchases) and the rui term is our original matrix of purchases."

It's just a way of scaling the preferences to make a positive implicit feedback much greater than a negative implicit feedback of zero. Please see the paper I linked to by Hu, Koren, and Volinsky for more information.

http://yifanhu.net/PUB/cf.pdf

• Reply • Share ›

王逸峰 Jesse Steinweg-Woods • a month ago

Get it~

Thank you soooooooo much!!!!!!!

• Reply • Share ›

David Lucey • 2 months ago

Hi Jesse,

Thanks so much for this post. I referred heavily to it in a recent Masters Capstone Project on another small online store, and now trying to replicate a similar one using Michael Hahsler's recommenderlab to fill a gap for other R users.

I'm struggling because my baseline popular AUC is coming in around 52%, and ALS implicit is

I'm struggling because my baseline popular AUC is coming in around 52%, and ALS implicit is also doing much more poorly at 59% (lambda =0.1, alpha=40, n_iterations=10, n_factors=20). Random is giving an AUC of 50% which makes me think that the data is prepared in the matrix correctly. I also tried using binary user-item feedback instead of using quantity purchased, but the results were also poor.

I have 3,659 items and 4,333 users with 266,223 transaction pairs in my matrix:

see more

^ | ⌄ • Reply • Share ›

Jesse Steinweg-Woods Mod ➦ David Lucey • 2 months ago

Hi David,

Well you are improving on the baseline some so I think the matrix factorization is working somewhat. My daughter was actually just born so I can't give you a more detailed response right now but I would guess your matrix is either too sparse or there is something wrong with the way you are setting up the user/item matrix. I would also try upping the number of iterations and/or number of factors. The hyperparameters I used may be a poor fit for your data.

^ | ⌄ • Reply • Share ›

David Lucey ➦ Jesse Steinweg-Woods • 2 months ago

Jesse, Many congratulations! I am using the same data as your were with the UCI UK online store. I have used recommenderlab successfully before and think the user-item matrix is set up properly as shown in the screen shots. The sparsity is going to be 1- 266,233 / 4333 * 3658 = 97.6%. The parameters of ALS should be the same. It is perplexing that popular comes out so much different. A long shot, but in case anything else comes to mind later on. Many thanks responding with such big events going on. David

^ | ⌄ • Reply • Share ›

Ankit • 2 months ago

Hi Jessie, in the implicit package documentation, it is stated that This matrix should be a csr_matrix where the rows of the matrix are the item, the columns are the users that liked that item, and the value is the confidence that the user liked the item. Here you have done the opposite. Does it affect the output?

^ | ⌄ • Reply • Share ›

Jesse Steinweg-Woods Mod ➦ Ankit • 2 months ago

Hi Ankit,

The version I used in the notebook is a much older version and at some point the author changed the ordering of things for reasons I don't understand (wish he hadn't because I get so many questions about it!)

It doesn't affect the output as long as the dimensions of the matrix line up properly when calculating the dot product of the latent user/item vectors (since essentially if I had reversed the item/user ordering it is just the transpose of the matrix).

^ | ˅ • Reply • Share ›

Agus Yudiana • 3 months ago

Hi Jessie, this is really an excellent article!! I've learned a lot from it, I even tried to write a code to evaluate the ALS model from implicit module, and it worked oke. however when I try to calculate using cosine similarity the calc_mean_auc function returned error because the CosineRecommender class has no item_factors and user_factor which I used to put in the prediction parameter in the calc_mean_auc function. do you have any idea how can I evaluate this cosine method using your calc_mean_auc function ?

^ | ˅ • Reply • Share ›

Jesse Steinweg-Woods Mod → Agus Yudiana • 3 months ago

Hi Agus,

You won't be able to use the function I wrote. You will have to write your own. The reason is because the recommendations returned by my method are essentially a dot product between the user's vector and all item vectors.

So I would just change the code to utilize a list of recommended items for each user and compare this to what items they actually purchased based on the function I wrote. You will have to be a bit creative but I think you can do it.

Good luck!

- Jesse

^ | ˅ • Reply • Share ›

Agus Yudiana → Jesse Steinweg-Woods • 3 months ago

Hi Jesse,

Thank you for the quick response, in my case i would like to start with the list of all user and its the top-N recommended item data which I produced and then utilize the information in product_users_altered to calculate the ROC. do you think it's possible ?

thanks a lot for the discussion I really appreciate it.

^ | ˅ • Reply • Share ›

Jesse Steinweg-Woods Mod → Agus Yudiana • 3 months ago

Yeah that should work. I would give it a try.

^ | ˅ • Reply • Share ›

Alessandro Terragni • 4 months ago

Hello Jessie, good article !

I have a question regarding the item_vecs and user_vecs

Using the new version of the implicit library in this way:

model = implicit.als.AlternatingLeastSquares(factor = 20, regularisation = 0.1, iterations = 50)
model.fit(product_train*alpha)
item_vecs = model.item_factors
user_vecs = model.user_factors

The function
calc_mean_auc(product_train, items_users_altered,
[sparse.csr_matrix (user_vecs), sparse.csr_matrix (item_vecs.T)],
product_test)
gives an error.

While swapping item_vecs and user_vecs gives the correct results.

Is there an error in the calc_mean_auc function ?

Thank you in advance

• Reply • Share ›

Jesse Steinweg-Woods Mod → Alessandro Terragni • 4 months ago

Hi Alessandro,

Glad you liked the article!

When I wrote this originally, the ordering of the returns was user vectors/item vectors. The author switched the order for some reason. This probably also switched the dimensions of the matrices. Check the dimensions of your user/item matrices and make sure the product will work (if you understand linear algebra). There isn't a bug with the code if the user matrix has dimensions of num_rows x rank and the items matrix has dimensions of num_columns x rank. If you transpose the items matrix in this case and take the product of the two, you get a final matrix with dimensions of num_rows x num_columns.

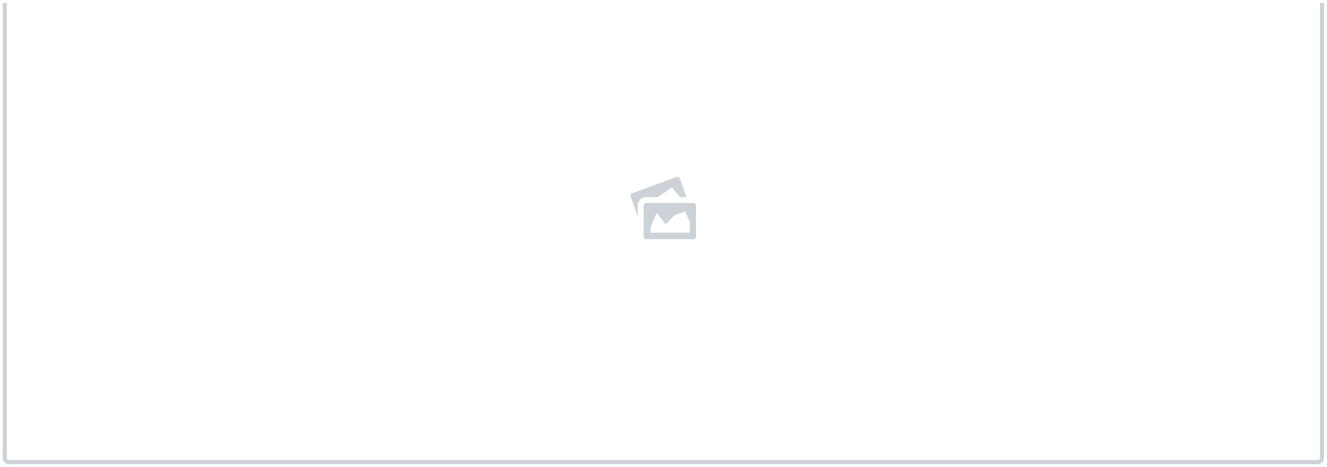
So with the original version of the library it works fine. You can try installing the earliest version and you will see it runs with no issues.

- Jesse

• Reply • Share ›

Flavia García • 4 months ago

Hi Jesse, thank you very much for your article, it's really helpful !! I'm executing your code and when I'm doing the ALS with the same number of factors I have this:



Also I don't understand why the alpha parameter.

Thank you in advanced!

^ | v • Reply • Share ›

Jesse Steinweg-Woods Mod ➦ Flavia García • 4 months ago

Hi Flavia,

My guess is your problem has to do with the changes to the implicit library compared to when I wrote this post. Ben Frederickson changed the API for the library so I would try using his updated code (documentation is here):

http://implicit.readthedocs...

See if that fixes your problem. Regrading the alpha parameter, see what I wrote in the post:

"The α term represents a linear scaling of the rating preferences (in our case number of purchases) and the rui term is our original matrix of purchases."

The original paper discusses that term is meant to be a linear scaling of the ratings preference matrix. This can allow the matrix factorization to give an interaction more weight than no interaction would otherwise (instead of having 1 and 0 in the matrix, you can have 15 and 0 or 40 and 0. This changes the factorization, giving higher weight to non-sparse entries).

Hope that helps!

- Jesse

^ | v • Reply • Share ›

Luis • 5 months ago

Hi Jesse, thank you very much for sharing this information, it is very useful.
My question is about hyper parameter tuning. I would like use a method such as sklearn's GridSearchCV, but the cross validation procedure this method uses works by removing selecting certain rows for training and certain rows for testing. Do you know how I could use your implementation of training/validation split with GridSearch?

If not, how did you select the hyper parameters? Or what is your approach for hyper parameter selection?

Thanks

^ | ˅ • Reply • Share ›

Jesse Steinweg-Woods Mod ➦ Luis • 5 months ago

Hi Luis,

I understand that desire. I don't think this would work with gridsearch (this isn't an sklearn model). You would just need to try some automated programming yourself (such as through a for loop) and test different hyperparameters (there are really only two: the regularization parameter and the rank size of the latent matrices for users/items.)

I just played around with a few settings and tried to see which improved the AUC most. This isn't the proper way to do it, however. You should have a train/validation/test set and only do hyperparameter tuning on the validation set (which could simply be k-fold cross-validation on the training set). I just simplified things to make this demonstration easier.

^ | ˅ • Reply • Share ›

Jacob Gaudoin • 6 months ago

Hi Jesse, thanks for the fantastic article. I'm just getting into this field and this was a great introduction to the topic.

I've looked around a bit and wasn't able to find a solution for how we can go about combining multiple types of implicit and explicit data into a single recommendation model. For example, the purchase quantity as described in this article (implicit), plus the number of page views that the item received from each user (implicit), plus the explicit 5 star rating given to the product by the user etc...

One thought I had was to create separate models for each type of data and feed those models into a "general" one if possible, but I don't know if there is a better solution? Would this even be likely to provide valuable results?

On a different note, I modified the code for obtaining the user_vecs and item_vecs vectors with implicit, as the function used in this article is deprecated. I thought I would provide this in case any other beginners like myself need a hand. (I'm not sure why users and items are reversed, but it worked that way and I didn't want to dig in to the implicit code right now to understand.)

alpha = 15
item_user_data = (product_train*alpha).astype('double')
model = implicit.als.AlternatingLeastSquares(factors=20, regularization = 0.1, iterations = 50)
model.fit(item_user_data)
user_vecs, item_vecs = model.item_factors, model.user_factors.transpose()

Thanks again!

^ | ˅ • Reply • Share ›

Jesse Steinweg-Woods Mod ➦ Jacob Gaudoin • 6 months ago

Hey Jacob,

Glad the article was of help!

For combining multiple pieces of information, if you want to stick to just pure matrix factorization there are methods you can TECHNICALLY do that I would not recommend, such as creating different user/item matrices and putting the values into each ratings matrix based on a specific filter.

Something you can do that works better is to use the matrix factorization from this example and grab the latent features for each user/item. Then combine these features with others you want to use and put them into a binary classification model you train. This version would require you to score each latent item feature to see the probability of it being purchased. You could then rank your recommendations based on the probability of being purchased.

Regarding the code changes, yes Ben Frederickson updated his implicit library. Why he switched the order of user_vecs/item_vecs I have no idea but he did.

• Reply • Share ›

Jacob Gaudoin → Jesse Steinweg-Woods • 6 months ago

Ok thanks, if I understood the "better" approach correctly:

1. Perform matrix factorization for each type of data I want to use (purchases, views, explicit 1-5 rating etc...) to produce several latent user/item feature vectors.

2. Feed these item and user features (and others) into a classifier and train in order to score the probability of a combination of features being purchased.

3. With the trained classifier, score every user/item pair.

4. Rank the recommendations based on the highest scoring recommendations.

How about managing new users/items? Does this mean we need to reperform multiple matrix factorizations to find the latent features for a new user/item? Although with some initial investigation, it seems that there are methods to get around this.

• Reply • Share ›

Jesse Steinweg-Woods Mod → Jacob Gaudoin • 6 months ago

That's the idea. For new users/items yes. If you don't have enough information on new users/items yet, you can try feeding these features into the model as missing, or just use an easier baseline (such as most popular) for the new users until you have collected enough data on them.

• Reply • Share ›

Josh Chua • 6 months ago

Thanks for the article, Jesse!

In Collaborative Filtering for Implicit Feedback Datasets by Hu, Koren, and Volinsky, the authors use Mean Percentage Rank in their evaluation. Why did you chose to use AUC instead of MPR? In your mind, what are the biggest tradeoffs?

• Reply • Share ›

Jesse Steinweg-Woods Mod → Josh Chua • 6 months ago

Good question Josh!

I don't have anything against using MPR. I just knew AUC could measure the ability of the recommender to rank an item the user actually bought against an item they didn't. I also knew it would work fine on heavily imbalanced classes (since users won't buy most items).

I feel there are too many metrics in the recsys literature and in the future I would probably just use mean average precision (MAP @ k) for some number k instead. The original paper argued they couldn't do that because:

" . . . we are currently
unable to track user reactions to our recommendations.
Thus, precision based metrics are not very appropriate,
as they require knowing which programs are undesired
to a user. "

I don't necessarily agree with that statement though.

Glad you enjoyed the article!

• Reply • Share ›

wikigo • 6 months ago

Hi, Jesse. This is a pretty good post, and I learned a lot. By the way, I am looking at a similar problem with unary implicit feedback but the rating is always 1 (similar to Facebook 'like' button). Do you consider the above technique still works for this problem? or do you have any suggestion for this kind of problem? Thanks

• Reply • Share ›

Jesse Steinweg-Woods Mod → wikigo • 6 months ago

Hey wikigo,

Yep you could definitely apply these techniques to your problem. If they give a like, that is a 1. Otherwise it is a zero if a user never liked the item. (Similar to purchased/not purchased an item).

• Reply • Share ›

Ricardo Castro • 7 months ago

Jesse, this is an AMAZING post, congratulations and thanks a lot!

• Reply • Share ›

Jesse Steinweg-Woods Mod → Ricardo Castro • 7 months ago

Thanks Ricardo glad you liked it!

• Reply • Share ›

mag • 9 months ago

Hi Jesse,
I would like to begin by saying its an awesome post. great write up and explanation. I am a data science student.
While trying out the above codes i found that the recommendation output after multiplying user and item factors should range from 0-1. i m getting few recommendations above 1

. Please shed some light

• Reply • Share ›

Jesse Steinweg-Woods Mod → mag • 9 months ago

Hi mag,

Sorry for the delay in getting back to you (slipped my mind!)

I'm glad you enjoyed the post. There is no guarantee of any kind that the dot products of user/item vectors will be between 0-1. That's pretty normal. What really matters is the rank order of the dot product. The greater the dot product, the more likely an item is predicted to be recommended. Just sort them from greatest to least to output your final list of recommendations.

Remember, the dot product is between the latent user/item vectors formed through alternating least squares. This is not normalized to between 0-1 (unless you want to do this yourself explicitly afterwards). If you look at the rec_items function, this DOES scale the recommendations between 0-1 so that the rank order will work properly when multiplying by zero everything already purchased.

Hope this helps you out.

• Reply • Share ›

mag → Jesse Steinweg-Woods • 9 months ago

Thanks a lot for getting back and clarifying.

.

I had another query, when u say "which I found in testing does the best" Can you please let me know what did you observe and on what basis did you finalist on alpha, regularization of for that matter number of factors.
As i am new to this kind of model building. Can you please shed some light. Any pointers would be helpful

• Reply • Share ›

Jesse Steinweg-Woods Mod → mag • 9 months ago

One thing you could do is play around with the hyperparameter settings and then see how the final AUC score is affected. That's how I did it.

Ideally you would have separate train/test/validation sets so you don't overfit. In this case we just have a train/test set so if you were actually building a model in the real world, you need to make sure that the test set remains completely untouched.

• Reply • Share ›

Rajiv Abraham • a year ago

Thank you for this lovely article. I had a question of your usage of the `implicit` Python library.
In this article, `user_vec` is first i.e. `

user_vecs, item_vecs = implicit.alternating_least_squares(..)`

However, the return values of `alternating_least_squares` is the reverse i.e. `model.item_factors, model.user_factors`
Ref: https://github.com/benfred/...

Is this a typo or maybe I understood it incorrectly?

• Reply • Share ›

Jesse Steinweg-Woods Mod → Rajiv Abraham • a year ago

No you are not wrong. It turns out the author changed the code after I wrote this post and refactored it (see here):

https://github.com/benfred/...

You may have to change things a little bit to deal with this refactoring.

• Reply • Share ›

Rajiv Abraham → Jesse Steinweg-Woods • a year ago

Yes, I just had to reverse the variables. Thanks for confirming!

^ | v • Reply • Share ›

Monu Chaudhary • a year ago

^ | v • Reply • Share ›

Monu Chaudhary ➦ Monu Chaudhary • a year ago

please help me with this problem...i tried running the same code u provided but came across this problem and unable to fix it...

^ | v • Reply • Share ›

Jesse Steinweg-Woods Mod ➦ Monu Chaudhary • a year ago

Hi Monu,

Get rid of the transpose on your item_vecs and it should work. Instead of:

calc_mean_auc(... sparse.csr_matrix(item_vecs.T)), do:

calc_mean_auc(... sparse.csr_matrix(item_vecs).

The reason for this is because it looks like you never actually used the implicit library and its alternating_least_squares function. This outputs the item vectors with dimensions that are the opposite of implicit_weighted_ALS (slow) method I created.

That should fix your problem.

^ | v • Reply • Share ›

Juan Rendon • a year ago

I try this code with other little dataset, but when i retrieve the items to be recommended, i get some items "interacted" before. How can i send you my dataset to try it?

^ | v • Reply • Share ›

Jesse Steinweg-Woods Mod → Juan Rendon • a year ago

Hi Juan,

I would suggest you look at the code inside of the rec_items function more carefully and make sure your input data is formatted correctly with a zero if the item hasn't been interacted with yet and a 1 otherwise.

• Reply • Share ›

Juan Rendon → Jesse Steinweg-Woods • a year ago

If the item not been interacted yet by the user i assing the value -1 (on the cvs dataset no interacted items with -1 value: https://github.com/juanremi... instead of 0 (dataset interacted items value 0: https://github.com/juanremi..., because if the SUM() is 0, the item will be considered as interacted and changue the value 0 to 1 (dataset without non interacted items https://github.com/juanremi...

BUT IF YOU DONT WANT INCLUDE THE INTERACTED ITEMS BY AN USER
- no is necesasy include the not interacted items on the cvs by an user, except if the user is new and no interact with any item (coldstart problem), this might cause when you try to get info of the user an error, because the user not exist. tha same occurs when an useris new and the interactions with this item value is -1, because no is considered and you get the same error

RESULTS in both cases:
- Sometimes in the purchased items, not appears all the products to by an user. For example if I have 14 products and one user buy 5 for each product, on the purchased items list not appears all the products, only appears 12 products and i if run again the code i get the same results.

see more

• Reply • Share ›

Jesse Steinweg-Woods Mod → Juan Rendon • a year ago

I don't think I will be able to help you (as this sounds too complex).

I would recommend you make sure your initial matrix is as similarly formatted to the example shown here as possible and debug from there.

For recommending user to user, I would use cosine similarity of the latent user vectors as a starting point (same with item to item).

For user to item (sort of unusual to do that), you could just look at the top rankings for the same dot product of latent user vectors/latent item vectors, then rank the top users for a specific item.

• Reply • Share ›

Juan Rendon → Jesse Steinweg-Woods • a year ago for example (as you can see on the image http://imgur.com/a/8WFXU). In the first table you can see all the available ratings that the users (names at first column) do to the products (names at first row)... so you says i need fill with "0" if the item hasn't been interacted (as you can see in the second table), ... so i have my dataset in csv and run the code i get the next: http://imgur.com/a/UcHME the first are the "interacted items" and the second after the xxxxxx are the recommended items.. so can u see in the recommended items are some items that appears in the "interacted items list".:(

Reply • Share ›

Load more comments

ALSO ON **JESSE'S WEBSITE**

A Guide to Gradient Boosted Trees with XGBoost in Python – Jesse …

31 comments • 2 years ago

Avatar ak102 — great tutorial, thanks! PS converting probabilities to binary can be done with: np.rint(y_pred)also doing …

Predicting Airline Delays – Jesse Steinweg-Woods – Ph.D. Student and Data Science …

7 comments • 3 years ago

Avatar mirza abbas — hi jesseby any chance have u done the data analysis part in the python ?rather than using r

Web Scraping Indeed for Key Data Science Job Skills – Jesse Steinweg-Woods – …

50 comments • 3 years ago

Avatar Anthony Flores — Yo litterally was just about to start working on this. haha! Could you msg me if have any success? Tryin to job hunt.EDIT: …

An Introduction to Deep Learning using nolearn – Jesse Steinweg-Woods – Ph.D. …

16 comments • 3 years ago

Avatar Jesse Steinweg-Woods — Hi Tej,You COULD do that, but this isn't the best library for that anymore.In the case we have before us, you …

Subscribe **Add Disqus to your siteAdd DisqusAdd** **Disqus' Privacy PolicyPrivacy PolicyPrivacy**