


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



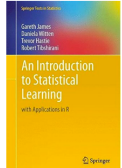
Python for Data Analysis:
Data Wrangling with Pandas, NumPy, and...
Wes McKinney
★★★★☆ 118
Paperback
\$27.68 ✓Prime



Data Science for Business:
What You Need to Know about Data Mining and...
Foster Provost & Tom Fawcett
★★★★☆ 135
Paperback
\$37.99 ✓Prime



Reproducible Research with R and RStudio,
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



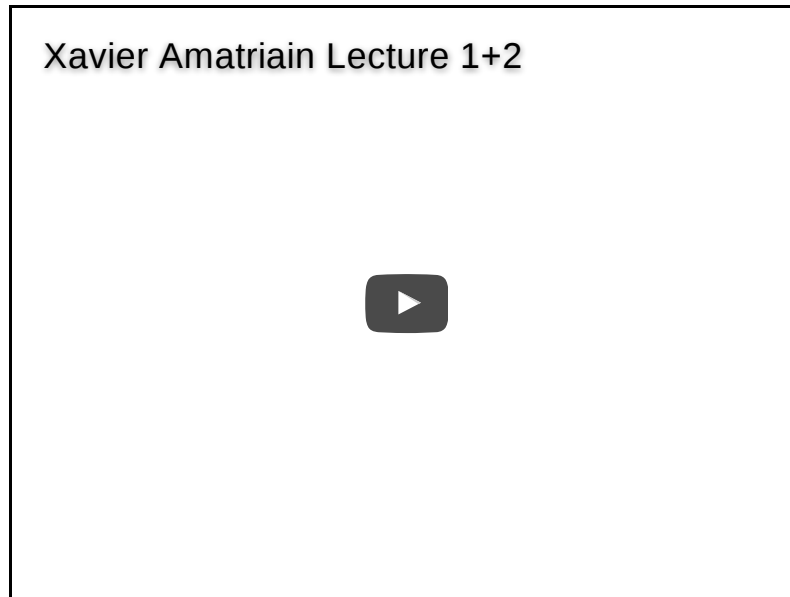
The Statistical Sleuth: A Course in Methods of Data Analysis
Fred Ramsey
★★★★☆ 6
Hardcover
\$284.42 ✓Prime

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

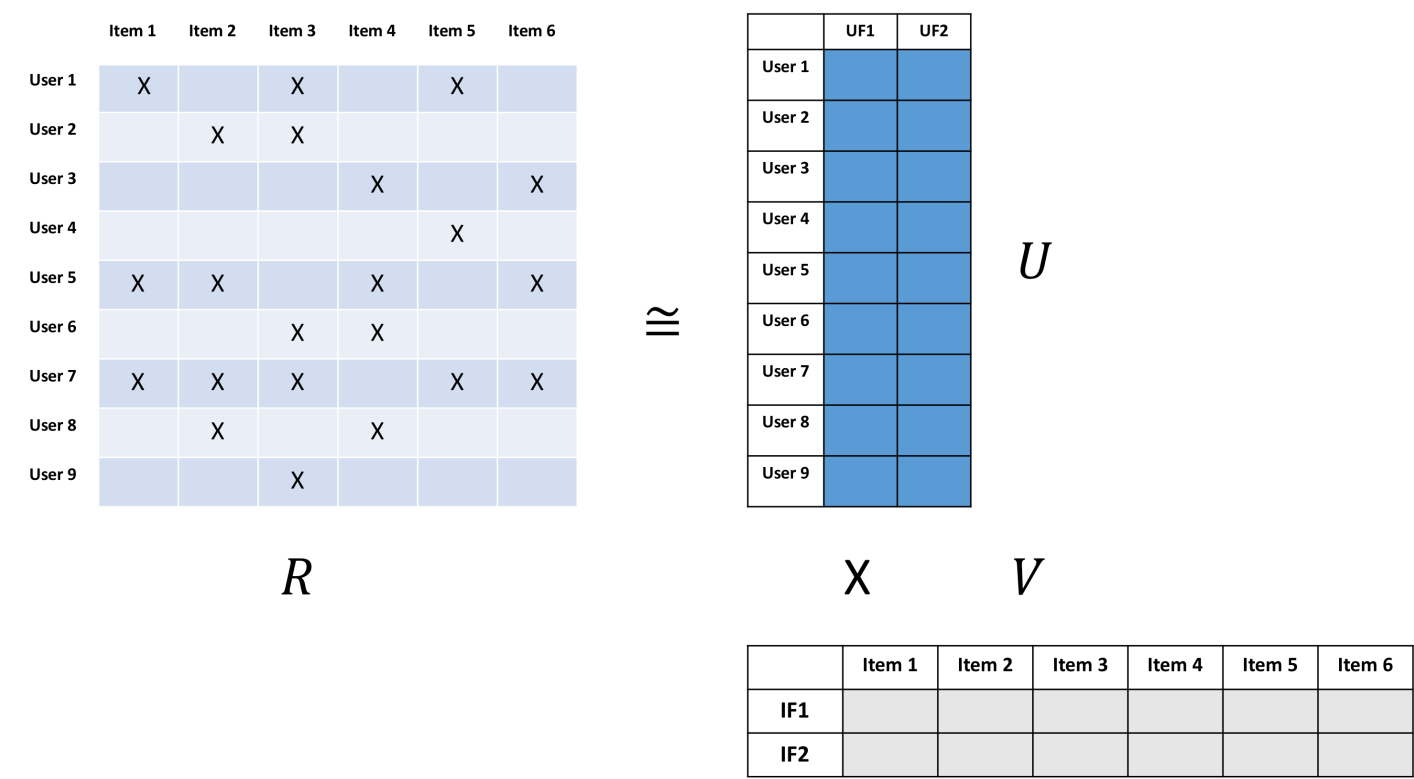
Before we start building our own recommender system on an example problem, I want to explain some of the intuition behind how this method works and why it likely is the only chosen method in Spark's library. We discussed before how collaborative filtering doesn't require any information about the users 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 $M \times N$, 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 $M \times K$ which will be our latent user feature vectors for each user (U) and a second with dimensions $K \times N$, 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/On
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
0536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	2010-12-01 08:26:00	2.55	17850.0	United Kingdom
1536365	71053	WHITE METAL6 LANTERN	6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom

InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
2536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	2010-12-01 08:26:00	2.75	17850.0	United Kingdom
3536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom
4536365	84029E	RED WOOLLY 6 HOTTIE WHITE HEART.		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() # 0
item_lookup['StockCode'] = item_lookup.StockCode.astype(str) # Encode as strings
```

```
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 t
cleaned_retail = cleaned_retail[['StockCode', 'Quantity', 'CustomerID']] # Get r
grouped_cleaned = cleaned_retail.groupby(['CustomerID', 'StockCode']).sum().rese
grouped_cleaned.Quantity.loc[grouped_cleaned.Quantity == 0] = 1 # Replace a sum
# indicate purchased
grouped_purchased = grouped_cleaned.query('Quantity > 0') # Only get customers w
```

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

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



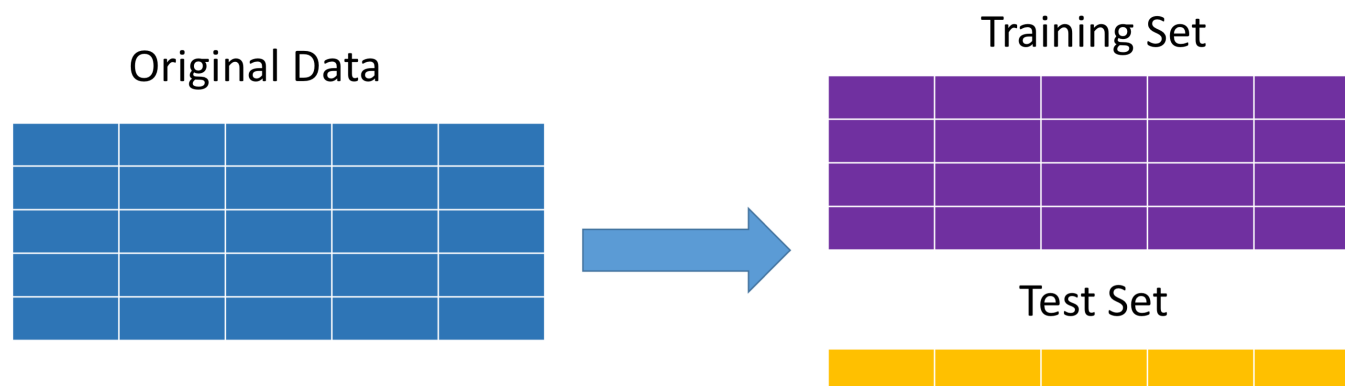
```
sparsity
```

```
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

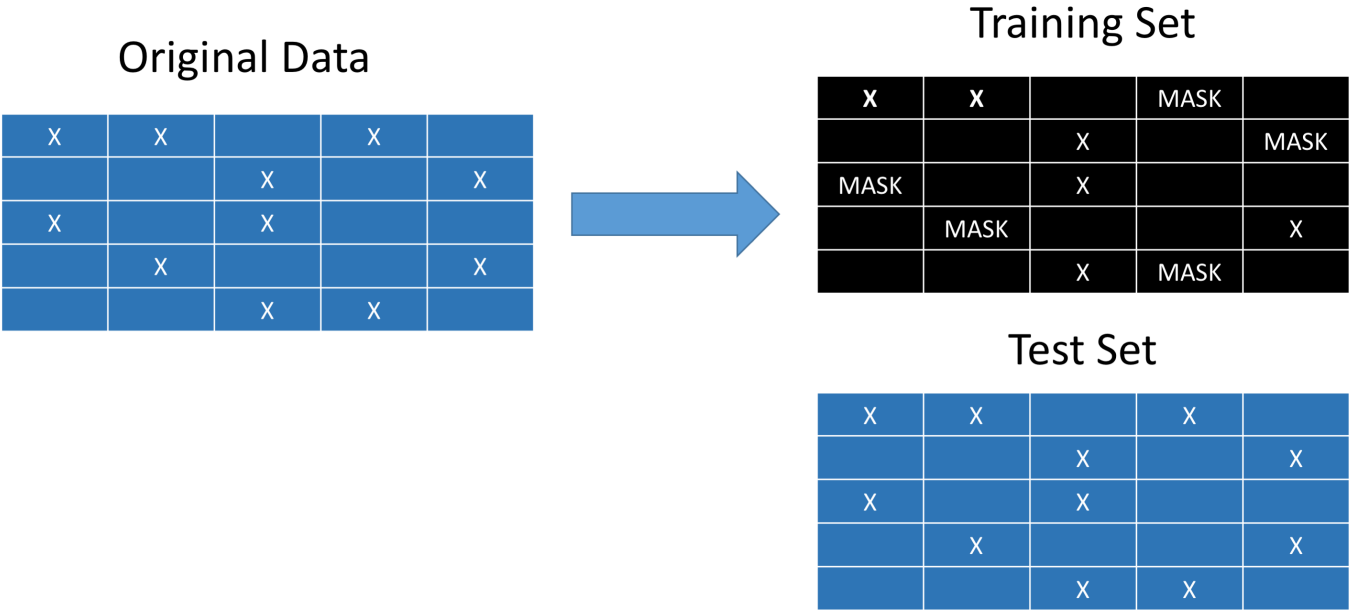
Typically in Machine Learning applications, we need to test whether the model we just trained is any good on new data it hasn't yet seen before from the training phase. We do this by creating a test set completely separate from the training set. Usually this is fairly simple: just take a random sample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

```
import random
```

```
def make_train(ratings, pct_test = 0.2):
```

```
    ...
```

```
    This function will take in the original user-item matrix and "mask" a percent  
    user-item interaction has taken place for use as a test set. The test set wi  
    while the training set replaces the specified percentage of them with a zero
```

```
    parameters:
```

`ratings` - the original ratings matrix from which you want to generate a training 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 too training set for later comparison to the test set, which contains all of the

returns:

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

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

`user_inds` - From the randomly selected user-item indices, which user rows we will use. 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
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
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 u
training_set.eliminate_zeros() # Get rid of zeros in sparse array storage af
return training_set, test_set, list(set(user_inds)) # Output the unique list
```

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 $Y^T Y$ portion without having to iterate through each user u . We can derive a similar equation for our items:

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

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
    feedback based collaborative filtering.

    parameters:

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

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

    alpha - The parameter associated with the confidence matrix discussed in the
    The paper found a default of 40 most effective. Decreasing this will decreas
```


various ratings.

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

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

seed - Set the seed for reproducible results

returns:

The feature vectors for users and items. The dot product of these feature vectors gives the predicted "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 a small value to the diagonal 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
Y = sparse.csr_matrix(rstate.normal(size = (num_item, rank_size))) # Normall
                                     # transpose at

X_eye = sparse.eye(num_user)
Y_eye = sparse.eye(num_item)
lambda_eye = lambda_val * sparse.eye(rank_size) # Our regularization term la

# We can compute this before iteration starts.

# Begin iterations

for iter_step in range(iterations): # Iterate back and forth between solving
    # Compute yTy and xTx at beginning of each iteration to save computing t
    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 matr
        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
```

```

yTCuIY = Y.T.dot(CuI).dot(Y) # This is the  $y^T(Cu-I)Y$  term
yTCuPu = Y.T.dot(CuI + Y_eye).dot(pref.T) # This is the  $y^TCuPu$  term,
                                           #  $Cu - I + I = Cu$ 

X[u] = spsolve(yTy + yTCuIY + lambda_eye, yTCuPu)
# Solve for  $Xu = ((yTy + y^T(Cu-I)Y + \lambda * I)^{-1})y^TCuPu$ , equation 4
# 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
    xTCiIX = X.T.dot(CiI).dot(X) # This is the  $x^T(Cu-I)X$  term
    xTCiPi = X.T.dot(CiI + X_eye).dot(pref.T) # This is the  $x^TCiPi$  term
    Y[i] = spsolve(xTx + xTCiIX + lambda_eye, xTCiPi)
    # Solve for  $Yi = ((xTx + x^T(Cu-I)X) + \lambda * I)^{-1}x^TCiPi$ , equation
# End iterations
return X, Y.T # Transpose at the end to make up for not being transposed at
               # 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, al  
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 embarrassingly 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).
                                                         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 me

parameters:

- predictions: your prediction output

- test: the actual target result you are comparing to

returns:

- AUC (area under the Receiver Operating Characteristic 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 thei
```

parameters:

training_set - The training set resulting from make_train, where a certain p
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 a
These should be stored in a list, with user vectors as item zero and item ve

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

test_set - The test set constucted earlier from make_train function

returns:

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

```
store_auc = [] # An empty list to store the AUC for each user that had an it  
popularity_auc = [] # To store popular AUC scores  
pop_items = np.array(test_set.sum(axis = 0)).reshape(-1) # Get sum of item i
```



```
item_vecs = predictions[1]
for user in altered_users: # Iterate through each user that had an item alte
    training_row = training_set[user,:].toarray().reshape(-1) # Get the trai
    zero_inds = np.where(training_row == 0) # Find where the interaction had
    # 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 originall
    actual = test_set[user,:].toarray()[0,zero_inds].reshape(-1)
    # Select the binarized yes/no interaction pairs from the original full d
    # that align with the same pairs in training
    pop = pop_items[zero_inds] # Get the item popularity for our chosen item
    store_auc.append(auc_score(pred, actual)) # Calculate AUC for the given
    popularity_auc.append(auc_score(pop, actual)) # Calculate AUC using most
# End users iteration

return float('%0.3f'%np.mean(store_auc)), float('%0.3f'%np.mean(popularity_auc)
# Return the mean AUC rounded to three decimal places for both test and popul
```

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)], pr  
# 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 mat
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, it
    '''
    This just tells me which items have been already purchased by a specific use

    parameters:

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

    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 des

    returns:
```

```
A list of item IDs and item descriptions for a particular customer that were
...

cust_ind = np.where(customers_list == customer_id)[0][0] # Returns the index
purchased_ind = mf_train[cust_ind,:].nonzero()[1] # Get column indices of pu
prod_codes = products_list[purchased_ind] # Get the stock codes for our purc
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_look
```

StockCode	Description
6161923166	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_l
    ...
    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 recommenda
    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
                    (in order of matrix)
    item_list - an array of the products that make up the columns of your rating
```

```
(in order of matrix)
```

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

```
num_items - The number of items you want to recommend in order of best recom
```

```
returns:
```

```
- The top n recommendations chosen based on the user/item vectors for items  
...
```

```
cust_ind = np.where(customer_list == customer_id)[0][0] # Returns the index  
pref_vec = mf_train[cust_ind,:].toarray() # Get the ratings from the trainin  
pref_vec = pref_vec.reshape(-1) + 1 # Add 1 to everything, so that items not  
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 use  
# 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 indi  
# 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 a
```

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_ar
          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

StockCode	Description
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_look
```

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_ar  
          num_items = 10)
```

StockCode	Description
022645	CERAMIC HEART FAIRY CAKE MONEY BANK
122055	MINI CAKE STAND HANGING STRAWBERRY
222644	CERAMIC CHERRY CAKE MONEY BANK
337447	CERAMIC CAKE DESIGN SPOTTED PLATE
437448	CERAMIC CAKE DESIGN SPOTTED MUG
522059	CERAMIC STRAWBERRY DESIGN MUG
622063	CERAMIC BOWL WITH STRAWBERRY DESIGN
722649	STRAWBERRY FAIRY CAKE TEAPOT
822057	CERAMIC PLATE STRAWBERRY DESIGN
922646	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_look
```

	StockCode	Description
34	22326	ROUND SNACK BOXES SET OF4 WOODLAND
35	22629	SPACEBOY LUNCH BOX

	StockCode	Description
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_ar  
          num_items = 10)
```

	StockCode	Description
022	662	LUNCH BAG DOLLY GIRL DESIGN
120	726	LUNCH BAG WOODLAND
220	719	WOODLAND CHARLOTTE BAG
322	383	LUNCH BAG SUKI DESIGN
420	728	LUNCH BAG CARS BLUE
523	209	LUNCH BAG DOILEY PATTERN
622	661	CHARLOTTE BAG DOLLY GIRL DESIGN
720	724	RED RETROSPOT CHARLOTTE BAG

StockCode	Description
823206	LUNCH BAG APPLE DESIGN
922384	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

We were unable to load Disqus. If you are a moderator please see our [troubleshooting guide](#).