Recommender systems
for clothing, shoes, and jewelry using Amazon data

User-based, item-based, and user/content hybrid methods of collaborative filtering

George Nakhleh & Gunjan Pandya

Abstract:

Amazon is arguably the company that brought recommender systems to the masses. “Users who like ___ also like ___” is a familiar phrase to anyone who has made purchases from the website, and recommender systems are a cornerstone of its business. For analysts interested in getting hands-on experience with recommender systems, then, Amazon is a familiar reference point. Fortunately, datasets of Amazon product reviews are available online for just this reason.

This project uses a subset of reviews for a specific set of Amazon products (Clothing, Shoes & Jewelry) to design different types of recommender systems. The end result is three separate methods of delivering product recommendations: user-based, item-based, and a user/content hybrid that leverages product metadata. These recommenders were evaluated using mean absolute error and recall.

[Topics: recommender systems, collaborative filtering, user-based, item-based, content-based, cosine similarity, mean absolute error, recall]

About the data

A brief summary of the dataset:

Clothing, Shoes & Jewelry: subset of Amazon clothing, shoes, and jewlery products that ensures each product has at least 5 reviews and each reviewer has reviewed at least 5 products.

Format: JSON, converted to csv for quicker input.

Source: University of California, San Diego (UCSD) (link)

Description: rows are reviews, variables described below:

- asin: unique identifier for product, essentially a product id
- helpful: List of two items - # of users who found review helpful ([2,5] = 2 of 5 users found this review helpful
- overall: user review of the product on a 5-star scale
- reviewText: text of the review
- reviewTime: date of the review (mm.dd.yyyy)
- reviewerID: ID of the user
- reviewerName: user name
- summary: summary of the review text
- unixReviewTime: time in unix format

Size: 278,677 rows, 9 columns (112 MB)

Snapshot of the dataset:

	asin	helpful	overall	reviewText	reviewTime	reviewerID	reviewerName	summary	unixReviewTime
0	0000031887	[0, 0]	5	This is a great tutu and at a really great pri...	02 12, 2011	A1KLRMWW2FWPL4	Amazon Customer "cameramom"	Great tutu- not cheaply made	1297468800
1	0000031887	[0, 0]	5	I bought this for my 4 yr old daughter for dan...	01 19, 2013	A2G5TCU2WDFZ65	Amazon Customer	Very Cute!!	1358553600
2	0000031887	[0, 0]	5	What can I say... my daughters have it in oran...	01 4, 2013	A1RLQXYNCMWRWN	Carola	I have buy more than one	1357257600
3	0000031887	[0, 0]	5	We bought several tutus at once, and they are ...	04 27, 2014	A8U3FAMSJVHS5	Caromcg	Adorable, Sturdy	1398556800
4	0000031887	[0, 0]	5	Thank you Halo Heaven great product for Little...	03 15, 2014	A3GEOILWLK86XM	CJ	Grammy's Angels Love it	1394841600

Metadata: additional information about each clothing, shoes, and jewelry product.
Description: rows are products, variables described below

- price: price of the item
- salesRank: where the product ranks in a given sales category
- categories: product categories that the given product falls under (list of lists)
- imUrl: link to an image of the product
- related: json-like data (dictionary) of similar items (eg ‘also viewed’, ‘also bought’)
- asin: product id
- title: name of the product
- brand: brand of the product

description: description of the product

Size: 1,503,384 rows, 9 columns (1.36 GB)

Snapshot of the dataset:

	price	salesRank	categories	imUrl	related	asin	title	brand	description
0	6.99	{'Clothing': 1233557}	[['Clothing, Shoes & Jewelry', 'Girls'], ['Clo...	http://ecx.images-amazon.com/images/I/31mCncNu...	{'also_viewed': ['B00JO8II76', 'B00DGN4R1Q', '...	0000037214	Purple Sequin Tiny Dancer Tutu Ballet Dance Fa...	Big Dreams	NaN
1	6.79	{'Sports & Outdoors': 8547}	[['Clothing, Shoes & Jewelry', 'Girls', 'Cloth...	http://ecx.images-amazon.com/images/I/314qZjYe...	{'also_bought': ['0000031852', '0000031895', '...	0000031887	Ballet Dress-Up Fairy Tutu	Boutique Cutie	This adorable basic ballerina tutu is perfect ...
2	64.98	{'Kitchen & Dining': 16987}	[['Clothing, Shoes & Jewelry', 'Novelty, Costu...	http://ecx.images-amazon.com/images/I/413tGhqo...	{'also_bought': ['B000BMTCK6', 'B0006JCGUM', '...	0123456479	SHINING IMAGE HUGE PINK LEATHER JEWELRY BOX / ...	NaN	Elegance par excellence. Hand-crafted of the f...
3	NaN	{'Clothing': 1180499}	[['Clothing, Shoes & Jewelry', 'Women', 'Acces...	http://ecx.images-amazon.com/images/I/31QZTHxv...	{'also_viewed': ['B008MTRT1O', 'B00BUG47S4', '...	0456844570	RiZ Women's Beautify Crafted ½ Rimmed F...	NaN	NaN

Exploratory analysis

Before setting up recommender systems and their prerequisite components, some time was spent sizing up the initial dataset. Below are some details of the data worth noting.

Average number of user reviews: The average user has reviewed 7 products (median of 6)
Average number of product reviews: The average product has 12 reviews (median: 8)
Avg product rating: 4.245, so reviewers are generally pretty generous.

Preprocessing

First, a user-item matrix needed to be generated, an array where rows are users and columns are each unique product, with values being the users' reviews of a given product. This way, we can see who reviewed what products how.

The user-item matrix is used to determine similarity between users and between products: users who review the same products the same way are considered similar, and products reviewed by the same users the same way are too. The ability to find most similar users (or products) to a given user (or product) is the basis of collaborative filtering.

The code below creates a user-item matrix, replacing null values with 0.

```
data_df_slim = data_df[['asin', 'overall', 'reviewerID']]

user_item_mat = data_df_slim.pivot(index='reviewerID', columns = 'asin', values='overall')

test = np.array(user_item_mat)

nonan_test = np.nan_to_num(test)
```

The resulting matrix has 39,387 rows and 23,033 columns. It is very sparse: Of the 23,033 possible products, keep in mind that the average user has reviewed around 7.

Snapshot of the user-item matrix, an array

```
In [24]: nonan_test

Out[24]: array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
                [ 0.,  0.,  0., ...,  0.,  0.,  0.],
                [ 0.,  0.,  0., ...,  0.,  0.,  0.],
                ...,
                [ 0.,  0.,  0., ...,  0.,  0.,  0.],
                [ 0.,  0.,  0., ...,  0.,  0.,  0.],
                [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
```

We can see above that we've moved into working with numpy arrays, and that the user-item matrix is very sparse.

At this point, we began implementing methods of collaborative filtering. As mentioned earlier, a first step in collaborative filtering methods for recommender systems is to determine most similar elements when given an element. In other words, if given a user, we find the most similar users to that one. Often this similarity measure is calculated for each user (or item) against all others, resulting in a similarity matrix.

Some measures of similarity handle sparsity better than others. In our case, we don't want two rows (users) who have reviewed none of the same items to be considered similar because both have so many products unrated (0's), this wouldn't make sense. Cosine similarity is a measure that is robust against this problem, 0's essentially don't count towards the resulting similarity, not ignored but effectively neutralized.

Since we are delivering recommendations based on similar users *or* similar items, we need to know the similarity of users to one another and items to one another. The result is two square matrices: a user similarity matrix and item similarity matrix. Row numbers and column numbers of user similarity matrix correspond to rows of the original data (users), and rows/columns of the item similarity matrix correspond to the column indices (products). This will be useful when we need to pull actual user reviews and get info on the products.

See the code below for generating user and item similarity matrices. Note that we implement the pairwise cosine similarity function from scikit-learn. Generating the item similarity matrix is as easy as transposing the user-item matrix and doing the same calculations.

```
#Creating user similarity matrix
user_sim_mat = metrics.pairwise.cosine_similarity(nonan_test)

#Creating item similarity matrix
nonan_test_t = nonan_test.T
itm_sim_mat = metrics.pairwise.cosine_similarity(nonan_test_t)
```

It should be noted that calculating similarity matrices can be time-intensive. To save time, we tried using a smaller function that only performs element-to-element similarity. This could be used in a loop to find the most similar element (eg user) to a given input element in a recommender function. This didn't actually save much time though.

See code below for determining cosine similarity:

```
#function to find similarity between two users
def find_sim(nonan_test,u1,u2) :

    cosine_sim = metrics.pairwise.cosine_similarity(nonan_test[u1], nonan_test[u2])# returns only a number
    return cosine_sim
```

For clarity, it makes sense to consider each collaborative filtering method separately.

User-based collaborative filtering

Once familiar with the steps, the process behind user-based collaborative filtering isn't too complex. In short, we're delivering our recommendations from products that similar users liked. In other words, given a user, we find similar users and take their word for it on what products are nice.

The first step is, given a user, finding users most similar to that user. This means finding users who reviewed the same products as the input user in the same ways. Next, we find what products these similar users reviewed. We now have a list of potential recommendations: of these products reviewed by similar users, which have the best rating among those users? These will be the products recommended to our input user.

We can control how many similar users to consider and how many recommendations to give.

See code below, that finds k most similar users:

```
#function to find k similar users based on precalculated user similarity matrix
def getUserPC(user_sim_mat,u,k):
    score=[]
    simUsers = []
    for others in range(len(user_sim_mat)):
        if others != u:
            score.append([user_sim_mat[u][others], others])
    score.sort()
    score.reverse()
    for item in score[:k]:
        simUsers.append(item[1])
    return simUsers
#return score[:k]
```

The code below will use getUserPC to find most similar users, then get their reviewed products (that the input user hasn't already reviewed), and deliver recommendations:

```python
#function to serve user based recommendations
def userBasedRecs(nonan_test, simUsers, unrated, numOfRecs, rated, unique_products, metadata):
    all_simusers_productsReviews = {}
    user_based_recs = []
    final_recs = []
    for user in simUsers:
        simUsers_productreviews = {}
        rated_items = np.where(nonan_test[user] != 0)[0]
        rated_item_reviews = np.extract(nonan_test[user] != 0, nonan_test[user])
        for i in list(zip(rated_items, rated_item_reviews)):
            for item in unrated:
                if item == i[0]:
                    simUsers_productreviews[i[0]] = i[1]
        #for item in unrated:
        #   if item not in simUsers_productreviews: del simUsers_productreviews[item]
        #print(simUsers_productreviews)
        for key in simUsers_productreviews:
            #If this product is not already in the master sim_users dictionary ...
            #... create a key-value pair for that product in the master list: make sure the value is a list
            if key not in all_simusers_productsReviews:
                all_simusers_productsReviews[key] = [simUsers_productreviews[key]]
            else:
                #If the user's product is already in the master sim_users dictionary
                #Find that product in the master dict and append the user's review to the existing list of reviews for that product
                all_simusers_productsReviews[key].append(simUsers_productreviews[key])
    #print(all_simusers_productsReviews)
    for key in all_simusers_productsReviews:
        user_based_recs.append([np.mean(all_simusers_productsReviews[key]), key])
    user_based_recs.sort()
    user_based_recs.reverse()
    for item in user_based_recs[:numOfRecs]:
        final_recs.append(item[1])

    #print(user_based_recs)
    #print(final_recs)

    all5 = []
    for item in user_based_recs:
        if item[0] == 5.0:
            all5.append(item[1])

    for item in rated:
        asin_pd = unique_products[item]
        product = metadata[metadata['asin']==asin_pd].title.item()# 'categories']]
        product = str(product)
        print("User Rated: " + product)
    print('\n')
    for item in final_recs:
        asin_pd = unique_products[item]
        product = metadata[metadata['asin']==asin_pd].title.item()# 'categories']]
        product = str(product)
        print("Recommendation: " + product)

    return final_recs
```

Let's see this in action.
Here is the output for a random user, serving 3 recommendations based on 5 most similar users

```
In [42]: # One large function: userBasedRecs
         reload(our_funcs)

         our_funcs.userBasedRecs(nonan_test, simUsers, unrated, num_recs, rated, unique_products, metadata)

User Rated: Embassy Italian Stone Design Genuine Leather Tote Bag
User Rated: Wall Earring Holder Jewelry Organizer Closet Jewelry Storage Rack - Earring Angel (Clear)
User Rated: Black Feather Boa
User Rated: UGG Australia Stain &amp; Water Repellent, One Bottle
User Rated: Sterling Silver Cubic Zirconia Medium Round Hoop Earrings
User Rated: Leg Avenue Wet Look Leggings with Elastic Lace up


Recommendation: EVogues Apparel Women's Plus Size Glitter Necklace Accented O-Ring Strap Top
Recommendation: Timberland PRO Men's Pitboss 6&quot; Steel-Toe Boot
Recommendation: Sterling Silver Multi-Color Amber Drop Earrings
```

Item-based collaborative filtering

What if, instead of getting recommendations from similar users, we took an approach of finding products similar to one we know we liked? That’s the intuition behind item-based collaborative filtering.

The steps are a bit shorter than user-based. First, we get an item that a given user likes,and calculated that item’s cosine similarity to all other products. After finding the most similar products, deliver ones the user hasn’t already reviewed as our recommendations. We don’t need to find rating information of these products since they are similar to one our user liked.

Use the item similarity matrix to find most similar items to a user’s top-reviewed product, and deliver recommendations based on the n most similar. See code below:

```
#Item-based recommender
#All-in-one: most similar items --> recommended items
def itemSim(nonan_test, u, num_recs, unique_products, metadata, itm_sim_mat, unRate, rate):


    #unRate, rate =  getUnrated(nonan_test,u)
    user_rev_list = []
    simItems = []
    #find their top rated item
    for prod in rate:
        user_rev = nonan_test[u,prod]
        user_rev_list.append([user_rev,prod])
        user_rev_list.sort()
        user_rev_list.reverse()

    most_liked_prod = user_rev_list[0][1]
    asin_pd = unique_products[most_liked_prod]
    product = metadata[metadata['asin']==asin_pd].title.item()# 'categories']]
    product = str(product)
    print("Most Liked Product: " + product+ '\n')

    score=[]
    simItems = []
    for others in range(len(itm_sim_mat)):
        if others != most_liked_prod:
            score.append([itm_sim_mat[most_liked_prod][others], others])
    score.sort()
    score.reverse()
    #print(score[:20])
    for item in score[:20]:
        if item[1] in rate:
            #print(item[1])
            score = list(filter((item).__ne__, score))
    #print(score)

    for item in score[:num_recs]:
        simItems.append(item[1])
    #print(simItems)
    for item in simItems:
        asin_pd = unique_products[item]
        product = metadata[metadata['asin']==asin_pd].title.item()# 'categories']]
        product = str(product)
        print("Recommended Product: " + product)
    #return score[:k]
```

Here is the output of the item-based recommender, serving 3 recommendations

In [43]:

```
#this function takes a user input, figures out the best-reviewed item by that user ...
#... then uses that best-reviewed-item

reload(our_funcs)

our_funcs.itemSim(nonan_test, u1, num_recs, unique_products, metadata)
```

```
Most Liked Product: Leg Avenue Wet Look Leggings with Elastic Lace up

Recommended Product: STEVEN by Steve Madden Women's Intyce Riding Boot
Recommended Product: Dunham by New Balance Men's St. Johnsbury Sandal
Recommended Product: 9884 Olive Drab Low Profile Death Spade Baseball Cap
```

Hybrid collaborative filtering

Often we have more context than just “who reviewed what how”. There’s information that is relevant to give personalized recommendations: information about the user or about the products: maybe a user reviews items from one brand very highly, or only buys items that have some specific feature (Kosher foods, for example). Simple review data could miss this.

In recommender systems, we can call recommender systems based on this information ‘content-based’. A common example is movie recommendations based on actors who appear in movies a user enjoyed, or based on genres the user watches most.

We used some product metadata to augment our user-based recommender system, resulting in a ‘hybrid’ of user and content-based collaborative filtering. Every product in the dataset has metadata related to product category. Naturally for our dataset, all fall under the Clothing, Shoes & Jewelry categories, but a pair of running shorts might also have categories like “Athletic wear” or “Women’s apparel”. Taking the categories for every product a user has reviewed, we can make a kind of ‘profile’ of each user. Next, after performing user-based collaborative filtering, finding products reviewed by similar users, we use the product categories of potentially recommended products and calculate the overlap between each product and the input user’s ‘profile’. In essence, we’re seeing if a product is in-line with the types of things the user buys.

For example, given a user, we find two similar users. These two users have reviewed 4 products that the user hasn’t. Two of them have an average rating of 5 stars among these 2 similar users. Our input user has bought products that have categories “Costume” “Costume jewelry” and “Clown shoes”. One of the items with 5 stars has “Costume” in common, while the other has none in common. This overlap acts as a weight on the reviews, punishing products that aren’t similar to the input user’s profile. This is a useful tiebreaker, then.

Code below:

Function for finding the category metadata of a product:

```python
#Function that takes the col indices of items rated by a user (aka one of the outputs of getUnrated) ...
# .. and returns a list of unique product categories (our user's 'content profile' for hybrid method)
def getCate(rated, unique_products, metadata):
    listn = []
    for item in rated:
        #print(item)
        asin_pd = unique_products[item]
        #print(asin_pd)
        cat = metadata[metadata['asin']==asin_pd].categories.item()# 'categories']]
        for i in cat:
            for j in i:
            #print(j)
                listn.append(j)
    unl = list(set(listn))
    return unl
```

Function for determining user-based recommendations, but before delivering, weighting by the overlap of each potentially recommended product's categories w/ the input user's category 'profile'

```python
#Hybrid recommender
def hybridRec(nonan_test, simUsers, unrated, numOfRecs, rated, unique_products, metadata):
    all_simusers_productsReviews = {}
    user_based_recs = []
    final_recs = []

    catUser = getCate(rated, unique_products, metadata)

    for user in simUsers:
        simUsers_productreviews = {}
        rated_items = np.where(nonan_test[user] != 0)[0]
        rated_item_reviews = np.extract(nonan_test[user] != 0, nonan_test[user])
        for i in list(zip(rated_items, rated_item_reviews)):
            for item in unrated:
                if item == i[0]:
                    simUsers_productreviews[i[0]] = i[1]
        #for item in unrated:
        #   if item not in simUsers_productreviews: del simUsers_productreviews[item]
        #print(simUsers_productreviews)
        for key in simUsers_productreviews:
            #If this product is not already in the master sim_users dictionary ...
            #... create a key-value pair for that product in the master list: make sure the value is a list
            if key not in all_simusers_productsReviews:
                all_simusers_productsReviews[key] = [simUsers_productreviews[key]]
            else:
                #If the user's product is already in the master sim_users dictionary
                #Find that product in the master dict and append the user's review to the existing list of reviews for that product
                all_simusers_productsReviews[key].append(simUsers_productreviews[key])
    #print(all_simusers_productsReviews)
    for key in all_simusers_productsReviews:
        user_based_recs.append([np.mean(all_simusers_productsReviews[key]), key])
    print("Original user-based recs:\n", user_based_recs)
    for item in user_based_recs:
        #print([item[1].item()])
        catItem = getCate([item[1].item()], unique_products, metadata)
        match = list(set(catUser).intersection(catItem))
        weight = len(match)/len(catItem)
        item[0] = item[0]*weight

    user_based_recs.sort()
    user_based_recs.reverse()
```

```python

for item in user_based_recs[:numOfRecs]:
    final_recs.append(item[1])

print("With content-based weighting:\n", final_recs)

for item in rated:
    asin_pd = unique_products[item]
    product = metadata[metadata['asin']==asin_pd].title.item()# 'categories']]
    product = str(product)
    print("User Rated:" + product)

for item in final_recs:
    asin_pd = unique_products[item]
    product = metadata[metadata['asin']==asin_pd].title.item()# 'categories']]
    product = str(product)
    print("Recommendation:" + product)

return final_recs
```

Here is output of the hybrid recommender system, 3 recommendations based on 5 most similar users:

```
In [45]: #Recommendations from Hybrid Recommender
         reload(our_funcs)

         our_funcs.hybridRec(nonan_test, simUsers, unrated, num_recs, rated, unique_products, metadata)
```

```
User Rated: Embassy Italian Stone Design Genuine Leather Tote Bag
User Rated: Wall Earring Holder Jewelry Organizer Closet Jewelry Storage Rack - Earring Angel (Clear)
User Rated: Black Feather Boa
User Rated: UGG Australia Stain &amp; Water Repellent, One Bottle
User Rated: Sterling Silver Cubic Zirconia Medium Round Hoop Earrings
User Rated: Leg Avenue Wet Look Leggings with Elastic Lace up


Recommendation: Bison Designs 38mm wide Light Duty Last Chance Belt with Gunmetal Buckle
Recommendation: Casio Men's AW49HE-2AV Ana-Digi Dual Time Watch
Recommendation: Sterling Silver Multi-Color Amber Drop Earrings
```

Evaluation

So we have delivered recommendations and we can kind of intuitively tell if they make sense or not, but what are standard, quantifiable ways of evaluating a recommender system?

Three common method are mean absolute error, precision, and recall. MAE is an evaluator of predictions (continuous) while precision and recall evaluate classifications (categorical), just as a point of distinction. We used MAE and recall to evaluate all three recommender systems that we designed.

Mean absolute error

Mean absolute error in the context of recommender systems means predicting how a user will review a product, and evaluating how far off our prediction was. Doing this many times and getting an average is the ‘mean’ in mean absolute error.

We find a product the input user has reviewed, withhold it as if it were unrated by the user, and depending on the method, find a way of predicting its rating.

For user-based, this involves finding the most similar users who have reviewed the withheld product, and getting their average rating. For item-based, this is a little more convoluted. We withhold one of the items that a given user has reviewed, then find the k most similar items *among the products reviewed by the user*. The user’s rating average of these products is what we would predict is given to the withheld product. MAE for the hybrid method is essentially the same as user-based MAE, except that we apply the content-based weighting to both the withheld item and the average reviews of the item from most similar users. In essence this just changes the scale, but the trends on error would be the same for both methods as you tweaked number of similar users.

See below, each method has both a base code for calculating absolute error between actual review and predicting review, and a function for performing this across a subset of the data and taking an average:

User-based

```python
#Given a user, find the most similar users who have also reviewed that user's 1st reviewed product ...
#Get the avg rating of that product from the k most similar users, that's our prediction
#Returns avg rating and actual rating ... this function is used for calculating MAE
def collabo_evaluator(user, k, nonan_test):

    simuser = []
    simu = []
    users_products = list(np.nonzero(nonan_test[user]))
    the_prod = users_products[0][0]  #arbitrary product from their list of products
    #print(the_prod)
    users_review = nonan_test[user, the_prod]
    #print(users_review)
       #find all users who have reviews 'the_prod'
    users_with_prod = np.where(nonan_test[:, the_prod] != 0)[0]
    for i in users_with_prod:
        if i != user:
            sim = find_sim(nonan_test,user,i)
            simuser.append([sim, i])
    simuser.sort()
    simuser.reverse()
    for item in simuser[:k]:
        simu.append(item[1])
    #print(simu)
    rat = 0
    for kuser in simu:
        rat += nonan_test[kuser][the_prod]
    pred_rating = rat/k
    #print(pred_rating)
    return pred_rating, users_review
```

Get MAE for user-based collaborative filtering on a subset of the data

```python
#performs many hybrid runs, using collabo_evaluator, returns mae
def calc_mae_user(nonan_test, ratio, k):

    #Get the subset of users
    num_users = np.shape(nonan_test)[0]
    list_of_users = list(range(0,num_users))
    np.random.shuffle(list_of_users)
    subset_size = int(num_users * ratio)
    subset_users = list_of_users[0:subset_size]
    #print(subset_users)
    abserror = 0
    count = 0
    for wuser in subset_users:
        pred, userr = collabo_evaluator(wuser, k, nonan_test)
        abserror += abs(pred-userr)
        #print(abserror)
        count += 1
    MAE = abserror/count
    print("MEAN ABSOLUTE ERROR: ", MAE)
```

Here’s the results when evaluation user-based recommender (5 most similar users) on 0.1% of the data:

```python
#User-based version of MAE

reload(our_funcs)

our_funcs.calc_mae_user(nonan_test, 0.01, n)
```

```
MEAN ABSOLUTE ERROR:  0.796946564885
```

Item-based

Calculating absolute error

```
#MAE for item-based
#We need to:
#Get a user's product reviews
#Pop one of their products and find most similar products to it (that the user has reviewed!)
#Get the average rating that the user has given to the most similar k items

def itemRec_evaluator(user, k, nonan_test, itm_sim_mat):

    score = []
    simuser = []
    simu = []
    simItems = []
    users_products = list(np.nonzero(nonan_test[user]))
    the_prod = users_products[0][0]  #arbitrary product from their list of products - popped product
    #print(the_prod)

    users_review = nonan_test[user, the_prod]
    #print(users_review)

    #find the products rated by user and its similarity to poped product
    unRate, rate =  getUnrated(nonan_test,user)
    #print(rate)
    for item in rate:
        if item != the_prod:
            score.append([itm_sim_mat[the_prod][item], item])
    score.sort()
    score.reverse()
    #print(score, '\n')
    for item in score[:k]:
        simItems.append(item[1])

    itmRatings = 0

    for item in simItems:
        itmRatings += nonan_test[user][item]
    pred_rating = itmRatings/k
    #print(pred_rating)
    #print(users_review)
    return pred_rating, users_review
```

Calculate MAE by performing the above function over a subset

```
def calc_mae_itm(nonan_test, ratio, k):

    #Get the subset of users
    num_users = np.shape(nonan_test)[0]
    list_of_users = list(range(0,num_users))
    np.random.shuffle(list_of_users)
    subset_size = int(num_users * ratio)
    subset_users = list_of_users[0:subset_size]
    #print(subset_users)
    abserror = 0
    count = 0
    for wuser in subset_users:
        pred, userr = itemRec_evaluator(wuser, k, nonan_test)
        abserror += abs(pred-userr)
        #print(abserror)
        count += 1
    MAE = abserror/count
    print("MEAN ABSOLUTE ERROR: ", MAE)
```

Hybrid

Calculating absolute error

```
def collabo_evaluator_hyb(user, k, nonan_test, unique_products, metadata):

    simuser = []
    simu = []
    rated_prod = np.where(nonan_test[user,:] != 0)[0]
    #print(rated_prod)
    users_products = list(np.nonzero(nonan_test[user])) #get products rated by user
    #print(users_products)
    #categories of products rated by user expect the product we are predicting for
    catUser = getCate(rated_prod[1:], unique_products, metadata)

    the_prod = users_products[0][0]                 #arbitrary product from their list of products
    users_review = nonan_test[user, the_prod]       #Users review for that product

    catItem = getCate([the_prod], unique_products, metadata)
    match = list(set(catUser).intersection(catItem))
    weight = len(match)/len(catItem)
    #print(weight)
    users_review = users_review*weight  #So we have weighted user review based on how similar that item is to all rated items
    #print(the_prod)

    #print(users_review)
    #find all users who have reviews 'the_prod'
    users_with_prod = np.where(nonan_test[:, the_prod] != 0)[0]
    for i in users_with_prod:
        if i != user:
            sim = find_sim(nonan_test,user,i)
            simuser.append([sim, i])
    simuser.sort()
    simuser.reverse()

    for item in simuser[:k]:
        simu.append(item[1])

    #print(simu)

    rat = 0
    othusers_review = 0
    #For every similar user, get their rating, and determine the average rating
    for kuser in simu:
        rat = nonan_test[kuser][the_prod]
        othrated_prod = np.where(nonan_test[kuser,:] != 0)[0]
        othrated_prod = list(filter((the_prod).__ne__, othrated_prod))
        othusers_products = list(np.nonzero(nonan_test[kuser]))  #list of products rated by similar user
        othusers_cat = getCate(othrated_prod, unique_products, metadata)
        match = list(set(othusers_cat).intersection(catItem))

        #print(match)
        #print(othusers_cat)
        #print(catItem)
        weight = len(match)/len(catItem)
        #print(weight)
        othusers_review += rat*weight

    pred_rating = othusers_review/k
    #print(pred_rating)
    #print(users_review)
    return pred_rating, users_review
```

Calculate MAE for hybrid method

```python
#performs many hybrid runs, using collabo_evaluator, returns mae
def calc_mae_hyb(nonan_test, ratio, k, unique_products, metadata):

    #Get the subset of users
    num_users = np.shape(nonan_test)[0]
    list_of_users = list(range(0,num_users))
    np.random.shuffle(list_of_users)
    subset_size = int(num_users * ratio)
    subset_users = list_of_users[0:subset_size]
    #print(subset_users)
    abserror = 0
    count = 0
    for wuser in subset_users:
        pred, userr = collabo_evaluator_hyb(wuser, k, nonan_test, unique_products, metadata)
        abserror += abs(pred-userr)
        #print(abserror)
        count += 1
    MAE = abserror/count
    print("MEAN ABSOLUTE ERROR: ", MAE)
```

Here are results when evaluating hybrid recommender (5 most similar users) on 1% of the data:

```python
#Hybrid version of MAE
reload(our_funcs)

our_funcs.calc_mae_hyb(nonan_test, 0.01, n, unique_products, metadata)
```

```
MEAN ABSOLUTE ERROR:  0.893930322129
```

Results

MAE, on 20% of the data (5% for hybrid method), adjusting parameters for number of similar users and number of products to recommend.

Note that this is taking a random subset of the data, so scores mildly fluctuate.

n: number of similar users/products to base recommendations off of

n=2	Method	ratio = 0.2 (0.05 for hybrid)
	User	0.7554
	Item	0.6938
	Hybrid	0.82168

n=5	Method	ratio = 0.2 (0.05 for hybrid)
	User	0.8859
	Item	0.7319
	Hybrid	0.8938

n=8	Method	ratio = 0.2 (0.05 for hybrid)
	User	1.1118
	Item	1.344
	Hybrid	1.08466

We see that MAE increases as we take more neighbors (either similar users or similar products). This makes sense, as we loosen the restriction on whose reviews can be considered when predicting a given user's review of a product, you'd think the less similar users would throw off the accuracy of prediction. One way of alleviating this, and improving the performance of the recommender systems in general, would be the weight the 'voice' a user is given in recommendations based on their similarity. The best performance is on item-based recommendation when the number of similar items is kept to a minimum.

Recall

Recall is about testing whether or not our recommendations are relevant. We hold out a product or several products that a given user liked, and we see if these products show up in our recommendations. Normally for our recommender systems, we wouldn't want to be serving up products the user has already purchased, so some additional work needs to be done to handle this.

Recall is then: (# of withheld products in recommendations) / (# of withheld products) . Because users have only reviewed a small number of products, we only withheld one product, and then repeated the process over a subset of the data to get an average recall for each recommender system.

Understanding how recall works, it becomes clear that there's a way to "game" the measure: if you serve up a large amount of recommendations, you are more likely to find the holdout product(s). That's worth keeping in mind: how the parameters will affect the measure, and what our priorities are (optimizing recall? Delivering a reasonable number of recommendations?).

See code below:

User-based

```
def user_based_recall(nonan_test, ratio, numSimUsers, numRecs,  unique_products, metadata, user_sim_mat):
    #get a subset of users
    num_users = np.shape(nonan_test)[0]
    list_of_users = list(range(0,num_users))
    np.random.shuffle(list_of_users)
    subset_size = int(num_users * ratio)
    subset_users = list_of_users[0:subset_size]

    user = 0
    user_recall = 0

    #for each index in that subset_users, pull the user at that row ...
    # ... and do process of getting similar users
    for wuser in subset_users:
        user+=1
        simUsers = getUserPC(user_sim_mat, wuser, numSimUsers)
        unrated, rated = getUnrated(nonan_test, wuser)
        wuser_reviews_list = []
        for product in rated:
            wuser_review = nonan_test[wuser, product]
            wuser_reviews_list.append([wuser_review, product])
        wuser_reviews_list.sort()
        wuser_reviews_list.reverse()

        #We now have the users product reviews and similar users.
        #We want to add validation prods to 'unrated' list, and serve recommendations to user
        validation_prods = wuser_reviews_list[0][1]
        #print(validation_prods)
        unrated.append(validation_prods)
        #print(unrated)
        recommendations = userBasedRecs(nonan_test, simUsers, unrated, numRecs, rated, unique_products, metadata)
        recommendations = map(int,recommendations)
        validation_prods = int(validation_prods)
        count = 0
        for rec in recommendations:
            if rec == validation_prods:
                count+=1
        user_recall += count/1
        #print(count)
        #print(user_recall)
        #print(user)

    final_recall = user_recall/user

    print(final_recall)
```

Item-based

```python
def item_based_recall(nonan_test, u, ratio, numSimUsers, numRecs, unique_products, metadata, itm_sim_mat):

    num_users = np.shape(nonan_test)[0]
    list_of_users = list(range(0,num_users))
    np.random.shuffle(list_of_users)
    subset_size = int(num_users * ratio)
    subset_users = list_of_users[0:subset_size]

    user = 0
    user_recall = 0
    for wuser in subset_users:
        user+=1
        unRate, rate =  getUnrated(nonan_test,wuser)
        user_rev_list = []

        #find their top rated item
        for prod in rate:
            user_rev = nonan_test[wuser,prod]
            user_rev_list.append([user_rev,prod])
            user_rev_list.sort()
            user_rev_list.reverse()
        print(user_rev_list)
        if len(user_rev_list) < 2:
            print("Can't perform recall!")
        else:
            most_liked_prod = user_rev_list[0][1]
            #validation_prod = user_rev_list[0][1]
            unRate.append(most_liked_prod)
            asin_pd = unique_products[most_liked_prod]
            product = metadata[metadata['asin']==asin_pd].title.item()# 'categories']]
            product = str(product)
            print("Most Liked Product:" + product)

            score=[]
            simItems = []

            for item in rate:
                if item != most_liked_prod:
                    #score.append([find_sim(nonan_test.T, most_liked_prod, item), item])
                    score.append([itm_sim_mat[most_liked_prod][item], item])
            score.sort()
            score.reverse()

            rate = []

            for item in score:
                rate.append(item[1])
            print(rate)
```

```python


            simItems = []

            prod_to_use = score[0][1]

            print(prod_to_use)

            score=[]

            for others in range(len(itm_sim_mat)):
                if others != prod_to_use:
                    score.append([itm_sim_mat[prod_to_use][others], others])
            score.sort()
            score.reverse()

            print(score[:20])
            for item in score[:20]:
                if item[1] in rate:
                    score = list(filter((item).__ne__, score))
    #print(score)

            for item in score[:numRecs]:
                simItems.append(item[1])
    #print(simItems)
            '''
            for item in simItems:
                asin_pd = unique_products[item]
                product = metadata[metadata['asin']==asin_pd].title.item()# 'categories']]
                product = str(product)
                print("Recommended Product: " + product)
                '''
            recommendations = map(int,simItems)
            validation_prods = int(most_liked_prod)
            count = 0
            for rec in recommendations:
                if rec == validation_prods:
                    count+=1
            user_recall += count/1
            print(count)
            print(user_recall)
            print(user)

    final_recall = user_recall/user

    print(final_recall)
```

Hybrid

```
#Hybrid based Recall
def hybrid_recall(nonan_test, ratio, numSimUsers, numRecs, unique_products, metadata, user_sim_mat):
    #get a subset of users
    num_users = np.shape(nonan_test)[0]
    list_of_users = list(range(0,num_users))
    np.random.shuffle(list_of_users)
    subset_size = int(num_users * ratio)
    subset_users = list_of_users[0:subset_size]

    user = 0
    user_recall = 0

    #for each index in that subset_users, pull the user at that row, and do process of getting similar users
    for wuser in subset_users:
        user+=1
        #simUsers = getUser(nonan_test, wuser, numSimUsers, similarity=find_sim)
        simUsers = getUserPC(user_sim_mat,wuser,numSimUsers)
        unrated, rated = getUnrated(nonan_test, wuser)
        wuser_reviews_list = []
        print("Items that user rated: ", rated)
        for product in rated:
            wuser_review = nonan_test[wuser, product]
            wuser_reviews_list.append([wuser_review, product])
        wuser_reviews_list.sort()
        wuser_reviews_list.reverse()

        #We now have the users product reviews and similar users.
        #We want to add validation prods to 'unrated' list, and serve recommendations to user
        validation_prods = wuser_reviews_list[0][1]
        print("Product pulled: ", validation_prods)
        unrated.append(validation_prods)
        #print(unrated)
        recommendations = hybridRec(nonan_test, simUsers, unrated, numRecs, rated, unique_products, metadata)
        recommendations = map(int,recommendations)
        validation_prods = int(validation_prods)
        count = 0
        for rec in recommendations:
            if rec == validation_prods:
                count+=1
        user_recall += count/1
        print(count)
        print(user_recall)
        print(user)

    final_recall = user_recall/user

    print("Final recall: ", final_recall)
```

Results

Recall, on 20% of the data (only 5% for hybrid method), adjusting parameters for number of similar users and number of recommendations

(note: Item-based recommendation is not affected by the number of similar users - in this implementation: number of recs would be synonymous with number of similar products to select. Variation is due to random sampling)

n: number of similar users, ratio: subset of the data to test on, num_recs: number of recommendations to deliver

n=2	**ratio = 0.2 (0.05 for hybrid)**	**num_recs = 2**	**num_recs = 5**	**num_recs = 8 (ratio = 0.1)**
	User	0.24423	0.37557	0.41705
	Item	0.25403	0.44527	0.57085
	Hybrid	0.31566	0.410138	0.41935

n=5	**ratio = 0.2 (0.05 for hybrid)**	**num_recs = 2**	**num_recs = 5**	**num_recs = 8**
	User	0.17289	0.34792	0.44124
	Item	0.24596	0.45506	0.58352
	Hybrid	0.28341	0.44009	0.49308

n=8	**ratio = 0.2 (0.05 for hybrid)**	**num_recs = 2**	**num_recs = 5**	**num_recs = 8**
	User	0.13594	0.26958	0.3899
	Item	0.25057	0.43894	0.57142
	Hybrid	0.20737	0.37557	0.45852

Analysis of results

From the results, we see the way recall can be “gamed”: we’ll be more likely to serve the held-out validation product as a recommendation as we increase the number of recommendations. Unfortunately, the recall here never even breaks the 50% threshold: more often than not, sometimes as often as ~70% of the time, the validation product is not in the recommendations.

On the other hand, this isn’t necessarily a great measure for our data: the dataset is very sparse, and the products are diverse. Especially for item-based recommendation, where hold out an item, then perform item-based recommendation on the product *most similar to the one held out*, how often will this methodology yield the results that recall is looking for? People buy a diverse range of clothing, shoes, and jewelry, unlike movies where we are at least confined to one format of products. If our recall function pulls out the only pair of shoes a user has reviewed, its likely that the most similar reviewed item, the one used to deliver recommendations, won't turn up that hold out product.

Conclusions

Our recommender systems were relatively successful in delivering relevant products to users. Given the sparsity of user data and the variety of products within the dataset, this encouraging. Mean absolute error was within a tolerable range for the most part (about 1-star off of the user's actual review), but when evaluated by recall the recommender systems are unimpressive. Whether or not the data even provides the opportunity for recall to be high (due to data sparsity), is an open question, but at the same time an "eyeball test" of how well the recommender systems are performing will not work at scale.

Two issues we encountered in implementing the three recommender systems and two evaluation methods were runtime and uncertainty on the best ways to leverage the metadata content provided with the data.

Runtime was especially an issue with the hybrid method. After reducing the metadata down to only those products which were in the reviews dataset, querying the metadata to retrieve product categories sped up greatly. Still, when running evaluation methods that have the recommender system performed hundreds of times, hybrid method was still very slow.

In the future, it would be interesting not only to leverage more of the metadata to make better content-based recommendations, but also to try an entirely different method of recommender systems from collaborative filter. For example, we did not explore matrix factorization in this project, and it would be worthwhile to see if that method brings any improvements to speed or accuracy/relevance of recommendations.