Recommendations with IBM

In this notebook, you will be putting your recommendation skills to use on real data from the IBM Watson Studio platform.

You may either submit your notebook through the workspace here, or you may work from your local machine and submit through the next page. Either way assure that your code passes the project RUBRIC. **Please save regularly.**

By following the table of contents, you will build out a number of different methods for making recommendations that can be used for different situations.

Table of Contents

At the end of the notebook, you will find directions for how to submit your work. Let's get started by importing the necessary libraries and reading in the data.

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import project_tests as t
import pickle

%matplotlib inline

df = pd.read_csv('user-item-interactions.csv')
df_content = pd.read_csv('articles_community.csv')
del df['Unnamed: 0']
del df_content['Unnamed: 0']
```

```
# Show df to get an idea of the data
df.head()
```

Out[ ]:

	article_id	title	email
0	1430.0	using pixiedust for fast, flexible, and easier...	ef5f11f77ba020cd36e1105a00ab868bbdbf7fe7
1	1314.0	healthcare python streaming application demo	083cbdfa93c8444beaa4c5f5e0f5f9198e4f9e0b
2	1429.0	use deep learning for image classification	b96a4f2e92d8572034b1e9b28f9ac673765cd074
3	1338.0	ml optimization using cognitive assistant	06485706b34a5c9bf2a0ecdac41daf7e7654ceb7
4	1276.0	deploy your python model as a restful api	f01220c46fc92c6e6b161b1849de11faacd7ccb2

In [ ]:
```
# Show df_content to get an idea of the data
df_content.head()
```

Out[ ]:

	doc_body	doc_description	doc_full_name	doc_status	article_id
0	Skip navigation Sign in SearchLoading...\r\n\r...	Detect bad readings in real time using Python ...	Detect Malfunctioning IoT Sensors with Streami...	Live	0
1	No Free Hunch Navigation * kaggle.com\r\n\r\n ...	See the forest, see the trees. Here lies the c...	Communicating data science: A guide to present...	Live	1
2	☰ * Login\r\n * Sign Up\r\n\r\n * Learning Pat...	Here's this week's news in Data Science and Bi...	This Week in Data Science (April 18, 2017)	Live	2
3	DATALAYER: HIGH THROUGHPUT, LOW LATENCY AT SCA...	Learn how distributed DBs solve the problem of...	DataLayer Conference: Boost the performance of...	Live	3
4	Skip navigation Sign in SearchLoading...\r\n\r...	This video demonstrates the power of IBM DataS...	Analyze NY Restaurant data using Spark in DSX	Live	4

Part I : Exploratory Data Analysis

Use the dictionary and cells below to provide some insight into the descriptive statistics of the data.

`1.` What is the distribution of how many articles a user interacts with in the dataset? Provide a visual and descriptive statistics to assist with giving a look at the number of times each user interacts with an article.

In [ ]:

In [ ]:

In [ ]:
```
# Fill in the median and maximum number of user_article interactios below

median_val = df.groupby(by='email')['article_id'].count().median() # 50% of individuals interact with ____ number of ar
max_views_by_user = df.groupby(by='email')['article_id'].count().max()# The maximum number of user-article interactions
```

`2.` Explore and remove duplicate articles from the **df_content** dataframe.

In [ ]:
```
# Find and explore duplicate articles
df_content.duplicated('article_id').sum()
```

Out[ ]: 5

In [ ]:
```
# Remove any rows that have the same article_id - only keep the first
df_content.drop_duplicates(subset='article_id',inplace=True)
```

`3.` Use the cells below to find:

a. The number of unique articles that have an interaction with a user.
b. The number of unique articles in the dataset (whether they have any interactions or not).
c. The number of unique users in the dataset. (excluding null values)
d. The number of user-article interactions in the dataset.

In [ ]:

In [ ]:
```
unique_articles = df.article_id.nunique()# The number of unique articles that have at least one interaction
total_articles = df_content.article_id.nunique()# The number of unique articles on the IBM platform
unique_users = df.email.nunique() # The number of unique users
user_article_interactions = int(df.groupby(by='email')['article_id'].count().sum())# The number of user-article interac
```

`4.` Use the cells below to find the most viewed **article_id**, as well as how often it was viewed. After talking to the company leaders, the `email_mapper` function was deemed a reasonable way to map users to ids. There were a small number of null values, and it was found that all of these null values likely belonged to a single user (which is how they are stored using the function below).

In [ ]:
```
most_viewed_article_id = str(df.groupby(by='article_id')['email'].count().sort_values(ascending=False).idxmax())# The mo
```

```
max_views =df.groupby(by='article_id')['email'].count().sort_values(ascending=False).iloc[0]# The most viewed article in
```

In [ ]:

```
## No need to change the code here - this will be helpful for later parts of the notebook
# Run this cell to map the user email to a user_id column and remove the email column

def email_mapper():
    coded_dict = dict()
    cter = 1
    email_encoded = []

    for val in df['email']:
        if val not in coded_dict:
            coded_dict[val] = cter
            cter+=1

        email_encoded.append(coded_dict[val])
    return email_encoded

email_encoded = email_mapper()
del df['email']
df['user_id'] = email_encoded

# show header
df.head()
```

Out[ ]:

	article_id	title	user_id
0	1430.0	using pixiedust for fast, flexible, and easier...	1
1	1314.0	healthcare python streaming application demo	2
2	1429.0	use deep learning for image classification	3
3	1338.0	ml optimization using cognitive assistant	4
4	1276.0	deploy your python model as a restful api	5

In [ ]:

```
unique_users = int(df.user_id.nunique())
user_article_interactions = df.groupby(by='user_id')['article_id'].count().sum()
```

In [ ]:

```
unique_users -=1
```

In [ ]:

```
## If you stored all your results in the variable names above,
```

```
## you shouldn't need to change anything in this cell

sol_1_dict = {
    '`50% of individuals have _____ or fewer interactions.`': median_val,
    '`The total number of user-article interactions in the dataset is ______.`': user_article_interactions,
    '`The maximum number of user-article interactions by any 1 user is ______.`': max_views_by_user,
    '`The most viewed article in the dataset was viewed _____ times.`': max_views,
    '`The article_id of the most viewed article is ______.`': most_viewed_article_id,
    '`The number of unique articles that have at least 1 rating ______.`': unique_articles,
    '`The number of unique users in the dataset is ______`': unique_users,
    '`The number of unique articles on the IBM platform`': total_articles
}

# Test your dictionary against the solution
t.sol_1_test(sol_1_dict)
```

```
It looks like you have everything right here! Nice job!
```

Part II: Rank-Based Recommendations

Unlike in the earlier lessons, we don't actually have ratings for whether a user liked an article or not. We only know that a user has interacted with an article. In these cases, the popularity of an article can really only be based on how often an article was interacted with.

`1.` Fill in the function below to return the **n** top articles ordered with most interactions as the top. Test your function using the tests below.

In [ ]:
```
def get_top_articles(n, df=df):
    '''
    INPUT:
    n - (int) the number of top articles to return
    df - (pandas dataframe) df as defined at the top of the notebook

    OUTPUT:
    top_articles - (list) A list of the top 'n' article titles

    '''
    top_articles = df.groupby(by='title')['user_id'].count().sort_values(ascending=False).index

    return top_articles[:n] # Return the top article titles from df (not df_content)
```

```
def get_top_article_ids(n, df=df):
    '''
    INPUT:
    n - (int) the number of top articles to return
    df - (pandas dataframe) df as defined at the top of the notebook

    OUTPUT:
    top_articles - (list) A list of the top 'n' article titles

    '''
    top_articles = df.groupby(by='article_id')['user_id'].count().sort_values(ascending=False).index

    return top_articles[:n] # Return the top article ids
```

In [ ]:
```
print(get_top_articles(10))
print(get_top_article_ids(10))
```

```
Index(['use deep learning for image classification',
       'insights from new york car accident reports',
       'visualize car data with brunel',
       'use xgboost, scikit-learn & ibm watson machine learning apis',
       'predicting churn with the spss random tree algorithm',
       'healthcare python streaming application demo',
       'finding optimal locations of new store using decision optimization',
       'apache spark lab, part 1: basic concepts',
       'analyze energy consumption in buildings',
       'gosales transactions for logistic regression model'],
      dtype='object', name='title')
Float64Index([1429.0, 1330.0, 1431.0, 1427.0, 1364.0, 1314.0, 1293.0, 1170.0,
              1162.0, 1304.0],
             dtype='float64', name='article_id')
```

In [ ]:
```
# Test your function by returning the top 5, 10, and 20 articles
top_5 = get_top_articles(5)
top_10 = get_top_articles(10)
top_20 = get_top_articles(20)

# Test each of your three lists from above
t.sol_2_test(get_top_articles)
```

```
Your top_5 looks like the solution list! Nice job.
Your top_10 looks like the solution list! Nice job.
Your top_20 looks like the solution list! Nice job.
```

Part III: User-User Based Collaborative Filtering

`1.` Use the function below to reformat the **df** dataframe to be shaped with users as the rows and articles as the columns.

- Each **user** should only appear in each **row** once.
- Each **article** should only show up in one **column**.
- **If a user has interacted with an article, then place a 1 where the user-row meets for that article-column**. It does not matter how many times a user has interacted with the article, all entries where a user has interacted with an article should be a 1.
- **If a user has not interacted with an item, then place a zero where the user-row meets for that article-column**.

Use the tests to make sure the basic structure of your matrix matches what is expected by the solution.

In [ ]:
```
# create the user-article matrix with 1's and 0's

def create_user_item_matrix(df):
    '''
    INPUT:
    df - pandas dataframe with article_id, title, user_id columns

    OUTPUT:
    user_item - user item matrix

    Description:
    Return a matrix with user ids as rows and article ids on the columns with 1 values where a user interacted with
    an article and a 0 otherwise
    '''
    user_item = df.groupby(['user_id','article_id'])['title'].count().unstack()
    user_item.fillna(0,inplace=True)



    def con(x):
        if x > 0 :
            return 1
        else:
            return 0
```

```
    lis = user_item.columns
    for i in lis:
        user_item[i] = user_item[i].apply(con)

    return user_item # return the user_item matrix

user_item = create_user_item_matrix(df)
```

In [ ]:
```
## Tests: You should just need to run this cell.  Don't change the code.
assert user_item.shape[0] == 5149, "Oops!  The number of users in the user-article matrix doesn't look right."
assert user_item.shape[1] == 714, "Oops!  The number of articles in the user-article matrix doesn't look right."
assert user_item.sum(axis=1)[1] == 36, "Oops!  The number of articles seen by user 1 doesn't look right."
print("You have passed our quick tests!  Please proceed!")
```

```
You have passed our quick tests!  Please proceed!
```

`2.` Complete the function below which should take a user_id and provide an ordered list of the most similar users to that user (from most similar to least similar). The returned result should not contain the provided user_id, as we know that each user is similar to him/herself. Because the results for each user here are binary, it (perhaps) makes sense to compute similarity as the dot product of two users.

Use the tests to test your function.

In [ ]:
```
def find_similar_users(user_id, user_item=user_item):
    '''
    INPUT:
    user_id - (int) a user_id
    user_item - (pandas dataframe) matrix of users by articles:
                1's when a user has interacted with an article, 0 otherwise

    OUTPUT:
    similar_users - (list) an ordered list where the closest users (largest dot product users)
                    are listed first

    Description:
    Computes the similarity of every pair of users based on the dot product
    Returns an ordered

    '''
    # compute similarity of each user to the provided user
    dic = {}
```

```
    for i in user_item.index :
        dot = np.dot(user_item.loc[user_id,:],user_item.loc[i,:])
        dic[i] = dot

    # sort by similarity
    sorted_dic = sorted(dic.items(), key=lambda x: x[1], reverse=True)


    # create list of just the ids
    most_similar_users = [i[0] for i in sorted_dic]

    # remove the own user's id
    most_similar_users.remove(user_id)

    return most_similar_users # return a list of the users in order from most to least similar
```

In [ ]:
```
# Do a spot check of your function
print("The 10 most similar users to user 1 are: {}".format(find_similar_users(1)[:10]))
print("The 5 most similar users to user 3933 are: {}".format(find_similar_users(3933)[:5]))
print("The 3 most similar users to user 46 are: {}".format(find_similar_users(46)[:3]))
```

```
The 10 most similar users to user 1 are: [3933, 23, 3782, 203, 4459, 131, 3870, 46, 4201, 49]
The 5 most similar users to user 3933 are: [1, 23, 3782, 203, 4459]
The 3 most similar users to user 46 are: [4201, 23, 3782]
```

`3.` Now that you have a function that provides the most similar users to each user, you will want to use these users to find articles you can recommend. Complete the functions below to return the articles you would recommend to each user.

In [ ]:
```
df['article_id'] = df['article_id'].astype(str)
def get_article_names(article_ids, df=df):
    '''
    INPUT:
    article_ids - (list) a list of article ids
    df - (pandas dataframe) df as defined at the top of the notebook

    OUTPUT:
    article_names - (list) a list of article names associated with the list of article ids
                    (this is identified by the title column)
    '''
    article_names = list(dict.fromkeys(df[df['article_id'].isin(article_ids)]['title']))
```

```
    return article_names # Return the article names associated with list of article ids


def get_user_articles(user_id, user_item=user_item):
    '''
    INPUT:
    user_id - (int) a user id
    user_item - (pandas dataframe) matrix of users by articles:
                1's when a user has interacted with an article, 0 otherwise

    OUTPUT:
    article_ids - (list) a list of the article ids seen by the user
    article_names - (list) a list of article names associated with the list of article ids
                    (this is identified by the doc_full_name column in df_content)

    Description:
    Provides a list of the article_ids and article titles that have been seen by a user
    '''
    ser = user_item.loc[user_id,:]
    article_ids = list(map(str,list(ser[ser == 1].reset_index()['article_id'])))

    article_names = get_article_names(article_ids)

    return article_ids, article_names # return the ids and names


def user_user_recs(user_id, m=10):
    '''
    INPUT:
    user_id - (int) a user id
    m - (int) the number of recommendations you want for the user

    OUTPUT:
    recs - (list) a list of recommendations for the user

    Description:
    Loops through the users based on closeness to the input user_id
    For each user - finds articles the user hasn't seen before and provides them as recs
    Does this until m recommendations are found

    Notes:
    Users who are the same closeness are chosen arbitrarily as the 'next' user
```

```
    For the user where the number of recommended articles starts below m
    and ends exceeding m, the last items are chosen arbitrarily

    '''
    similar_users = find_similar_users(user_id)


    watched_articles = []
    for i in similar_users:
        watched_articles.append(get_user_articles(i)[0])

    watched_articles = sum(watched_articles,[])
    watched_articles = dict.fromkeys(watched_articles)

    arr = dict.fromkeys(get_user_articles(user_id)[0])
    watched_articles = list(dict.fromkeys(i for i in watched_articles if i not in arr ))


    recs = watched_articles[:m]





    return recs # return your recommendations for this user_id
```

In [ ]:

```
# Check Results
get_article_names(user_user_recs(1, 10)) # Return 10 recommendations for user 1
```

Out[ ]:

```
['got zip code data? prep it for analytics. – ibm watson data lab – medium',
 'timeseries data analysis of iot events by using jupyter notebook',
 'graph-based machine learning',
 'using brunel in ipython/jupyter notebooks',
 'experience iot with coursera',
 'the 3 kinds of context: machine learning and the art of the frame',
 'deep forest: towards an alternative to deep neural networks',
 'this week in data science (april 18, 2017)',
 'higher-order logistic regression for large datasets',
 'using machine learning to predict parking difficulty']
```

In [ ]:

```
# Test your functions here - No need to change this code - just run this cell
assert set(get_article_names(['1024.0', '1176.0', '1305.0', '1314.0', '1422.0', '1427.0'])) == set(['using deep learning
assert set(get_article_names(['1320.0', '232.0', '844.0'])) == set(['housing (2015): united states demographic measures
```

```
assert set(get_user_articles(20)[0]) == set(['1320.0', '232.0', '844.0'])
assert set(get_user_articles(20)[1]) == set(['housing (2015): united states demographic measures', 'self-service data pr
assert set(get_user_articles(2)[0]) == set(['1024.0', '1176.0', '1305.0', '1314.0', '1422.0', '1427.0'])
assert set(get_user_articles(2)[1]) == set(['using deep learning to reconstruct high-resolution audio', 'build a python
print("If this is all you see, you passed all of our tests!  Nice job!")
```

```
If this is all you see, you passed all of our tests!  Nice job!
```

4. Now we are going to improve the consistency of the **user_user_recs** function from above.

- Instead of arbitrarily choosing when we obtain users who are all the same closeness to a given user - choose the users that have the most total article interactions before choosing those with fewer article interactions.
- Instead of arbitrarily choosing articles from the user where the number of recommended articles starts below m and ends exceeding m, choose articles with the articles with the most total interactions before choosing those with fewer total interactions. This ranking should be what would be obtained from the **top_articles** function you wrote earlier.

In [ ]:
```
def get_top_sorted_users(user_id, df=df, user_item=user_item):
    '''
    INPUT:
    user_id - (int)
    df - (pandas dataframe) df as defined at the top of the notebook
    user_item - (pandas dataframe) matrix of users by articles:
            1's when a user has interacted with an article, 0 otherwise


    OUTPUT:
    neighbors_df - (pandas dataframe) a dataframe with:
                    neighbor_id - is a neighbor user_id
                    similarity - measure of the similarity of each user to the provided user_id
                    num_interactions - the number of articles viewed by the user - if a u

    Other Details - sort the neighbors_df by the similarity and then by number of interactions where
                    highest of each is higher in the dataframe

    '''
    # getting similar users
    dic = {}
    for i in user_item.index :
        dot = np.dot(user_item.loc[user_id,:],user_item.loc[i,:])
        dic[i] = dot
```

```
    dic.pop(user_id)

    #converting dictionary into a dataframe
    neighbors_df = pd.DataFrame.from_dict(dic,orient='index')
    neighbors_df.rename(columns={0:'similarity'},inplace=True)
    neighbors_df['num_interactions'] = df.groupby('user_id')['article_id'].count()

    # sorting by similarity and number of articles viewed
    neighbors_df = neighbors_df.sort_values(by=['similarity','num_interactions'],ascending=False)

    return neighbors_df # Return the dataframe specified in the doc_string




def top_sorted_articles(article_ids,df=df):
    '''
    INPUT:
    article_ids : the article_ids that have been interacted with by similar users
    df - (pandas dataframe) df as defined at the top of the notebook

    OUTPUT:
    sorted_articles : sorted articles sorted by number of interactions

    '''
    df2 = df[df['article_id'].isin(article_ids)]
    sorted_articles = df2.groupby(by='title')['user_id'].count().sort_values(ascending=False).index

    return sorted_articles


def user_user_recs_part2(user_id, m=10):
    '''
    INPUT:
    user_id - (int) a user id
    m - (int) the number of recommendations you want for the user

    OUTPUT:
    recs - (list) a list of recommendations for the user by article id
    rec_names - (list) a list of recommendations for the user by article title

    Description:
```

```
    Loops through the users based on closeness to the input user_id
    For each user - finds articles the user hasn't seen before and provides them as recs
    Does this until m recommendations are found

    Notes:
    * Choose the users that have the most total article interactions
    before choosing those with fewer article interactions.

    * Choose articles with the articles with the most total interactions
    before choosing those with fewer total interactions.

    '''
    similar_users = get_top_sorted_users(user_id)


    watched_articles = []
    for i in similar_users.index:
        watched_articles.append(get_user_articles(i)[0])

    watched_articles = sum(watched_articles,[])
    watched_articles = dict.fromkeys(watched_articles)

    arr = dict.fromkeys(get_user_articles(user_id)[0])
    watched_articles = list(dict.fromkeys(i for i in watched_articles if i not in arr ))


    sorted_articles = top_sorted_articles(watched_articles)


    recs = watched_articles[:m]
    rec_names = get_article_names(recs)


    return recs, rec_names
```

In [ ]:

```
# Quick spot check - don't change this code - just use it to test your functions
rec_ids, rec_names = user_user_recs_part2(20, 10)
print("The top 10 recommendations for user 20 are the following article ids:")
print(rec_ids)
print()
print("The top 10 recommendations for user 20 are the following article names:")
print(rec_names)
```

```
The top 10 recommendations for user 20 are the following article ids:
['12.0', '109.0', '125.0', '142.0', '164.0', '205.0', '302.0', '336.0', '362.0', '465.0']

The top 10 recommendations for user 20 are the following article names:
['timeseries data analysis of iot events by using jupyter notebook', 'dsx: hybrid mode', 'accelerate your workflow with dsx', 'learn tensorflow and deep learning together and now!', "a beginner's guide to variational methods", 'tensorflow quick tips', 'challenges in deep learning', 'neural networks for beginners: popular types and applications', 'statistics for hackers', 'introduction to neural networks, advantages and applications']
```

`5.` Use your functions from above to correctly fill in the solutions to the dictionary below. Then test your dictionary against the solution. Provide the code you need to answer each following the comments below.

In [ ]:
```python
### Tests with a dictionary of results

user1_most_sim = get_top_sorted_users(1).index[0]
user131_10th_sim = get_top_sorted_users(131).index[9]
```

In [ ]:
```python
## Dictionary Test Here
sol_5_dict = {
    'The user that is most similar to user 1.': user1_most_sim,
    'The user that is the 10th most similar to user 131': user131_10th_sim,
}

t.sol_5_test(sol_5_dict)
```

```
This all looks good!  Nice job!
```

`6.` If we were given a new user, which of the above functions would you be able to use to make recommendations? Explain. Can you think of a better way we might make recommendations? Use the cell below to explain a better method for new users.

I would use **get_top_articles** to make him new recommendations , because I have no past data on him to know his prefrences. So I have to bet on that the best articles that most users interact with will be intersting to him too. And I think this is the best way until some data about his prefrences is collected.

`7.` Using your existing functions, provide the top 10 recommended articles you would provide for the a new user below. You can test your function against our thoughts to make sure we are all on the same page with how we might make a recommendation.

In [ ]:
```python
new_user = '0.0'

# What would your recommendations be for this new user '0.0'?  As a new user, they have no observed articles.
```

```
# Provide a list of the top 10 article ids you would give to
new_user_recs = list(map(str,get_top_article_ids(10)))
```

In [ ]:
```
assert set(new_user_recs) == set(['1314.0','1429.0','1293.0','1427.0','1162.0','1364.0','1304.0','1170.0','1431.0','1330

print("That's right!  Nice job!")
```

```
That's right!  Nice job!
```

Part IV: Content Based Recommendations (EXTRA - NOT REQUIRED)

Another method we might use to make recommendations is to perform a ranking of the highest ranked articles associated with some term. You might consider content to be the **doc_body**, **doc_description**, or **doc_full_name**. There isn't one way to create a content based recommendation, especially considering that each of these columns hold content related information.

`1.` Use the function body below to create a content based recommender. Since there isn't one right answer for this recommendation tactic, no test functions are provided. Feel free to change the function inputs if you decide you want to try a method that requires more input values. The input values are currently set with one idea in mind that you may use to make content based recommendations. One additional idea is that you might want to choose the most popular recommendations that meet your 'content criteria', but again, there is a lot of flexibility in how you might make these recommendations.

This part is NOT REQUIRED to pass this project. However, you may choose to take this on as an extra way to show off your skills.

In [ ]:
```
def make_content_recs():
    '''
    INPUT:

    OUTPUT:

    '''
```

`2.` Now that you have put together your content-based recommendation system, use the cell below to write a summary explaining how your content based recommender works. Do you see any possible improvements that could be made to your function? Is there anything novel about your content based recommender?

This part is NOT REQUIRED to pass this project. However, you may choose to take this on as an extra way to show off your skills.

Write an explanation of your content based recommendation system here.

`3.` Use your content-recommendation system to make recommendations for the below scenarios based on the comments. Again no tests are provided here, because there isn't one right answer that could be used to find these content based recommendations.

This part is NOT REQUIRED to pass this project. However, you may choose to take this on as an extra way to show off your skills.

In [ ]:
```
# make recommendations for a brand new user


# make a recommendations for a user who only has interacted with article id '1427.0'
```

Part V: Matrix Factorization

In this part of the notebook, you will build use matrix factorization to make article recommendations to the users on the IBM Watson Studio platform.

`1.` You should have already created a **user_item** matrix above in **question 1** of **Part III** above. This first question here will just require that you run the cells to get things set up for the rest of **Part V** of the notebook.

In [ ]:
```
# Load the matrix here
user_item_matrix = pd.read_pickle('user_item_matrix.p')
```

In [ ]:
```
# quick look at the matrix
user_item_matrix.head()
```

Out[ ]:

article_id	0.0	100.0	1000.0	1004.0	1006.0	1008.0	101.0	1014.0	1015.0	1016.0	...	977.0	98.0	981.0	984.0	985.0	986.0	990.0	9
user_id																			
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0	0.0	0.0	0.0	
2	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	
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	1.0	0.0	0.0	0.0	0.0	0.0	0.0	
4	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	
5	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	

5 rows × 714 columns

`2.` In this situation, you can use Singular Value Decomposition from numpy on the user-item matrix. Use the cell to perform SVD, and explain why this is different than in the lesson.

In [ ]:

```
# Perform SVD on the User-Item Matrix Here

u, s, vt = np.linalg.svd(user_item_matrix)
```

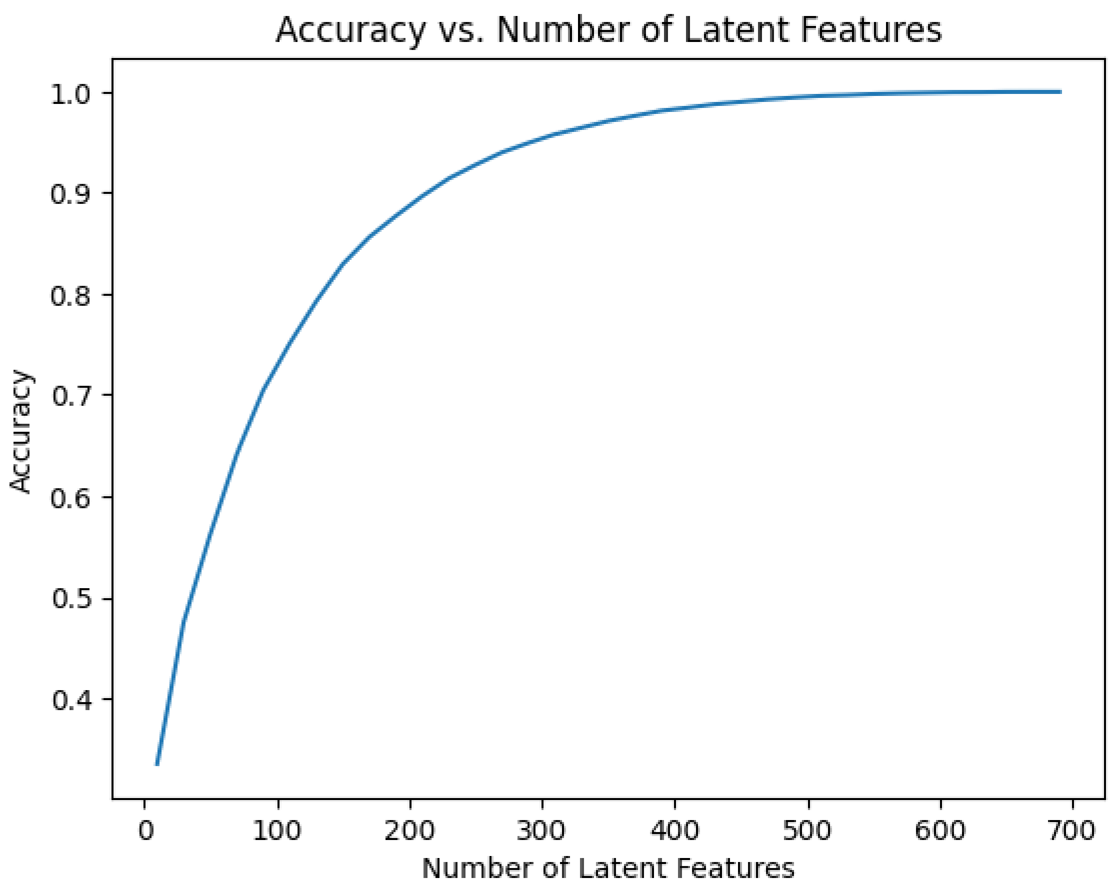
because we imputed the null values with zeros unlike the lesson , when we encountered null values we implemented funksvd

`3.` Now for the tricky part, how do we choose the number of latent features to use? Running the below cell, you can see that as the number of latent features increases, we obtain a lower error rate on making predictions for the 1 and 0 values in the user-item matrix. Run the cell below to get an idea of how the accuracy improves as we increase the number of latent features.

In [ ]:

```
num_latent_feats = np.arange(10,700+10,20)
sum_errs = []

for k in num_latent_feats:
    # restructure with k latent features
    s_new, u_new, vt_new = np.diag(s[:k]), u[:, :k], vt[:k, :]

    # take dot product
    user_item_est = np.around(np.dot(np.dot(u_new, s_new), vt_new))

    # compute error for each prediction to actual value
```

```
    diffs = np.subtract(user_item_matrix, user_item_est)

    # total errors and keep track of them
    err = np.sum(np.sum(np.abs(diffs)))
    sum_errs.append(err)


plt.plot(num_latent_feats, 1 - np.array(sum_errs)/df.shape[0]);
plt.xlabel('Number of Latent Features');
plt.ylabel('Accuracy');
plt.title('Accuracy vs. Number of Latent Features');
```



`4.` From the above, we can't really be sure how many features to use, because simply having a better way to predict the 1's and 0's of the matrix doesn't exactly give us an indication of if we are able to make good recommendations. Instead, we might split our dataset into a training and test set of data, as shown in the cell below.

Use the code from question 3 to understand the impact on accuracy of the training and test sets of data with different numbers of latent features. Using the split below:

- How many users can we make predictions for in the test set?
- How many users are we not able to make predictions for because of the cold start problem?
- How many articles can we make predictions for in the test set?
- How many articles are we not able to make predictions for because of the cold start problem?

In [ ]:
```
df_train = df.head(40000)
df_test = df.tail(5993)

def create_test_and_train_user_item(df_train, df_test):
    '''
    INPUT:
    df_train - training dataframe
    df_test - test dataframe

    OUTPUT:
    user_item_train - a user-item matrix of the training dataframe
                      (unique users for each row and unique articles for each column)
    user_item_test - a user-item matrix of the testing dataframe
                    (unique users for each row and unique articles for each column)
    test_idx - all of the test user ids
    test_arts - all of the test article ids

    '''
    #create_test_and_train_user_item
    user_item_train = create_user_item_matrix(df_train)
    user_item_test = create_user_item_matrix(df_test)

    test_idx = user_item_test.index
    test_arts = user_item_test.columns


    return user_item_train, user_item_test, test_idx, test_arts

user_item_train, user_item_test, test_idx, test_arts = create_test_and_train_user_item(df_train, df_test)
```

In [ ]:
```
#count how many users can we make predictions for in the test set
user_item_train.index.isin(test_idx).sum()
```

Out[ ]: 20

In [ ]:
```
#count How many users are we not able to make predictions for because of the cold start problem?
len(test_idx) - user_item_train.index.isin(test_idx).sum()
```

Out[ ]: 662

In [ ]:
```
#count how many articles can we make predictions for in the test set
user_item_train.columns.isin(test_arts).sum()
```

Out[ ]: 574

In [ ]:
```
#count How many articles are we not able to make predictions for because of the cold start problem?
len(test_arts) - user_item_train.columns.isin(test_arts).sum()
```

Out[ ]: 0

In [ ]:
```
# Replace the values in the dictionary below
a = 662
b = 574
c = 20
d = 0


sol_4_dict = {
    'How many users can we make predictions for in the test set?': c,
    'How many users in the test set are we not able to make predictions for because of the cold start problem?': a,
    'How many articles can we make predictions for in the test set?': b,
    'How many articles in the test set are we not able to make predictions for because of the cold start problem?': d
}

t.sol_4_test(sol_4_dict)
```

```
Awesome job!  That's right!  All of the test articles are in the training data, but there are only 20 test users that w
ere also in the training set.  All of the other users that are in the test set we have no data on.  Therefore, we canno
t make predictions for these users using SVD.
```

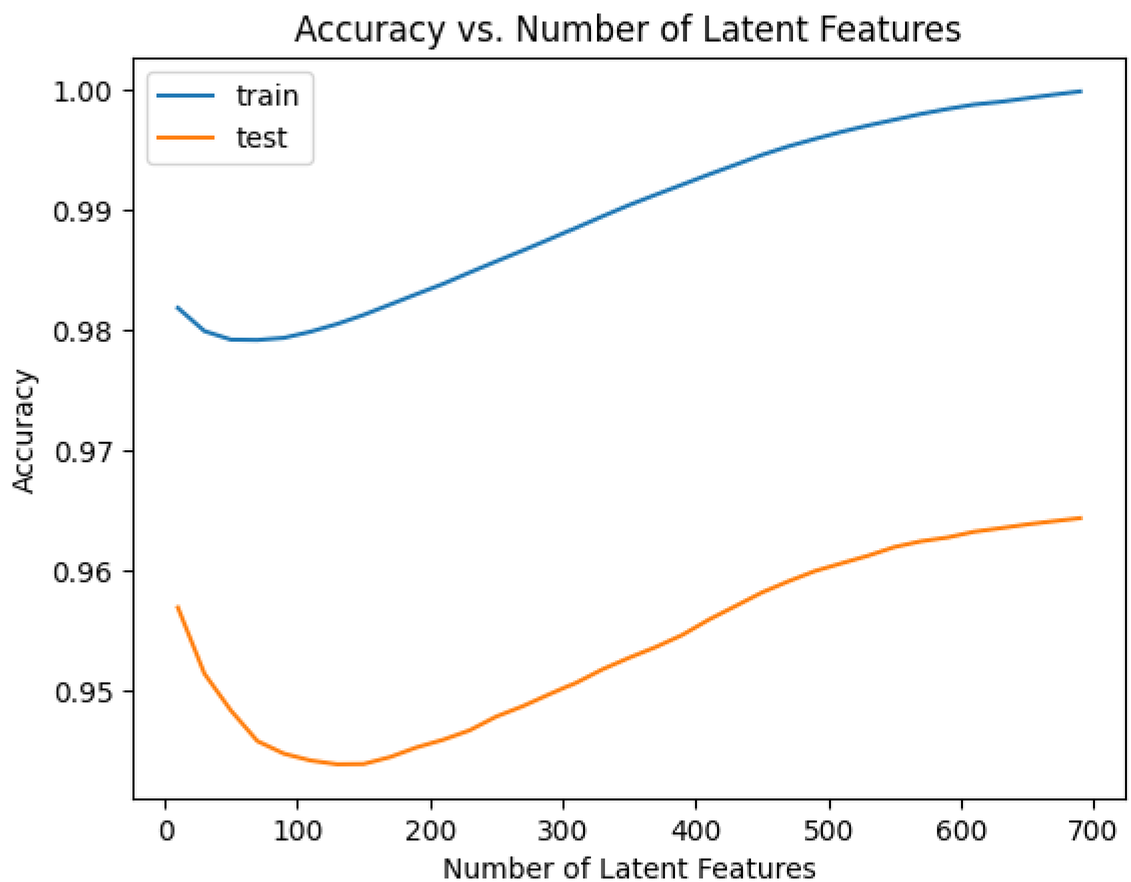
`5.` Now use the **user_item_train** dataset from above to find U, S, and V transpose using SVD. Then find the subset of rows in the **user_item_test** dataset that you can predict using this matrix decomposition with different numbers of latent features to see how many features makes sense to keep based on the accuracy on the test data. This will require combining what was done in questions `2` - `4`.

Use the cells below to explore how well SVD works towards making predictions for recommendations on the test data.

In [ ]:
```
# fit SVD on the user_item_train matrix
u_train, s_train, vt_train = np.linalg.svd(user_item_train)
```

In [ ]:
```
# Use these cells to see how well you can use the training
# decomposition to predict on test data
ids = user_item_train[user_item_train.index.isin(test_idx)].index
arts = user_item_train.iloc[:,user_item_train.columns.isin(test_arts)].columns

latent_features = np.arange(10,700+10,20)
sum_errors_tr = []
sum_error_ts = []

#finding the subset of rows in the test data
u_test = u_train[user_item_train.index.isin(test_idx),:]
vt_test = vt_train[:,user_item_train.columns.isin(test_arts)]

for feature in latent_features:
    #adjust the training data
    u_train_f = u_train[:,:feature]
    s_train_f = np.diag(s_train)[:feature,:feature]
    vt_train_f = vt_train[:feature,:]

    #adjust the test data
    u_test_f = u_test[:,:feature]
    vt_test_f = vt_test[:feature,:]

    #making predictions
    pred_train = np.dot(np.dot(u_train_f,s_train_f),vt_train_f)
    pred_test = np.dot(np.dot(u_test_f,s_train_f),vt_test_f)

    #calculating error
    sum_errors_tr.append((np.sum(np.sum(np.abs((user_item_train-pred_train))))))
    sum_error_ts.append((np.sum(np.sum(np.abs((user_item_test.loc[ids,arts]-pred_test))))))
```

In [ ]:
```
plt.plot(latent_features, 1 - np.array(sum_errors_tr)/(pred_train.shape[0]*pred_train.shape[1]),label='train');
plt.plot(latent_features, 1- np.array(sum_error_ts)/(pred_test.shape[0]*pred_test.shape[1]),label='test')
plt.xlabel('Number of Latent Features');
plt.ylabel('Accuracy');
plt.title('Accuracy vs. Number of Latent Features');
plt.legend();
```



`6.` Use the cell below to comment on the results you found in the previous question. Given the circumstances of your results, discuss what you might do to determine if the recommendations you make with any of the above recommendation systems are an improvement to how users currently find articles?

it seems from the above results the best number of latent features to keep is **690** but **10** latent features will not be a bad choice either considering the error and performance trade off but that is only applicape on the test data , And the downgrade of accuracy in the test data might be because of small number of users.

As to what might be done to know if our recommendation system makes a better difference for the user , We will have to make an experiment , where we use cookie based diversion to track users , And use invariant metrics as the count of the two groups those who will not recieve recommendations and those who will must be around the same

And our evaluation metric might be the rate of user-article interaction and if it proved to be statistically significant then we should apply the recommendations to all users .

Extras

Using your workbook, you could now save your recommendations for each user, develop a class to make new predictions and update your results, and make a flask app to deploy your results. These tasks are beyond what is required for this project. However, from what you learned in the lessons, you certainly capable of taking these tasks on to improve upon your work here!

Conclusion

> Congratulations! You have reached the end of the Recommendations with IBM project!

> **Tip**: Once you are satisfied with your work here, check over your report to make sure that it is satisfies all the areas of the rubric. You should also probably remove all of the "Tips" like this one so that the presentation is as polished as possible.

Directions to Submit

> Before you submit your project, you need to create a .html or .pdf version of this notebook in the workspace here. To do that, run the code cell below. If it worked correctly, you should get a return code of 0, and you should see the generated .html file in the workspace directory (click on the orange Jupyter icon in the upper left).

> Alternatively, you can download this report as .html via the **File** > **Download as** submenu, and then manually upload it into the workspace directory by clicking on the orange Jupyter icon in the upper left, then using the Upload button.

> Once you've done this, you can submit your project by clicking on the "Submit Project" button in the lower right here. This will create and submit a zip file with this .ipynb doc and the .html or .pdf version you created. Congratulations!

In [ ]:
```
from subprocess import call
```

```
call(['py', '-m', 'nbconvert', 'Recommendations_with_IBM.ipynb'])
```

Out[ ]: 1