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 libraries
import pandas as pd
import numpy as np
import random
import matplotlib.pyplot as plt
import project_tests as t
import pickle
import seaborn as sns

%matplotlib inline
```

In [ ]:
```
#load data
df = pd.read_csv('data/user-item-interactions.csv')
df_content = pd.read_csv('data/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 [ ]:

```
#convert article id from float to string
df.article_id = df.article_id.astype(str)
```

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 [ ]:
```
df.email.value_counts().describe()
```

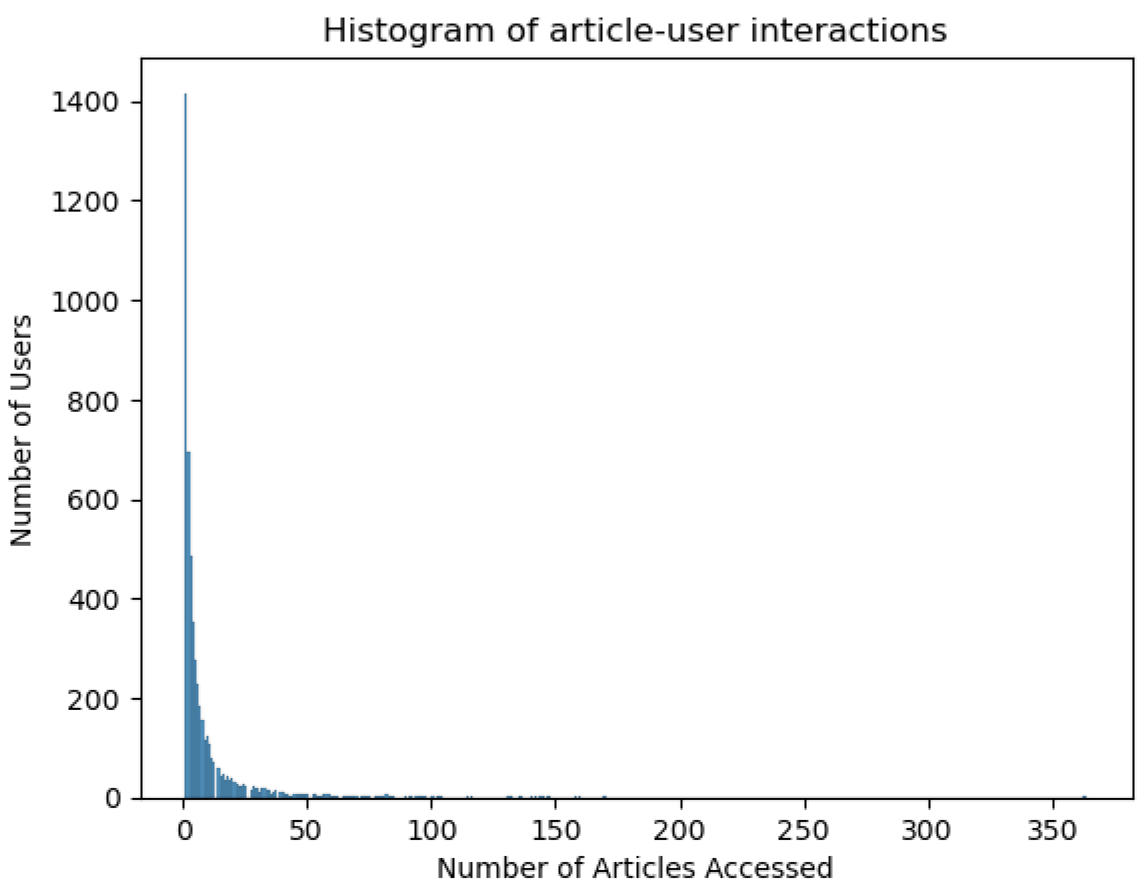
Out[ ]:
```
count    5148.000000
mean        8.930847
std        16.802267
min         1.000000
25%         1.000000
50%         3.000000
75%         9.000000
max       364.000000
Name: count, dtype: float64
```

In [ ]:
```
df.email.value_counts().median()
```

Out[ ]:
```
3.0
```

There are 5,148 users in the data, with the median user interacting with 3 articles. The distribution has mean 8.93, min 1, max 364, and sd 16.80.

In [ ]:
```
articles_per_user = df.email.value_counts()
p = sns.histplot(x=articles_per_user)
p.set(ylabel='Number of Users', xlabel='Number of Articles Accessed', title=
```



As we might expect, the vast majority of users access only a few articles and the distribution falls off sharply as the number of articles increases.

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

median_val = df.email.value_counts().median() # 50% of individuals interact
max_views_by_user = df.email.value_counts().max() # The maximum number of us
```

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

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

Out[ ]:
```
article_id
221     2
232     2
50      2
398     2
577     2
       ..
357     1
358     1
359     1
360     1
1050    1
Name: count, Length: 1051, dtype: int64
```

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

`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 [ ]:
```
unique_articles = len(df.article_id.unique()) # The number of unique article
total_articles = len(df_content.article_id.unique()) # The number of unique
unique_users = len(df.email.value_counts()) # The number of unique users
user_article_interactions = len(df.index) # The number of user-article inter
```

`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.article_id.value_counts().axes[0].tolist()[0
max_views = df.article_id.value_counts().max() # The most viewed article in
```

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

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	email	user_id
0	1430.0	using pixiedust for fast, flexible, and easier...	ef5f11f77ba020cd36e1105a00ab868bbdbf7fe7	1
1	1314.0	healthcare python streaming application demo	083cbdfa93c8444beaa4c5f5e0f5f9198e4f9e0b	2
2	1429.0	use deep learning for image classification	b96a4f2e92d8572034b1e9b28f9ac673765cd074	3
3	1338.0	ml optimization using cognitive assistant	06485706b34a5c9bf2a0ecdac41daf7e7654ceb7	4
4	1276.0	deploy your python model as a restful api	f01220c46fc92c6e6b161b1849de11faacd7ccb2	5

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 ______
    '`The maximum number of user-article interactions by any 1 user is _____
    '`The most viewed article in the dataset was viewed _____ times.`': max_
    '`The article_id of the most viewed article is ______.`': most_viewed_ar
    '`The number of unique articles that have at least 1 rating ______.`': u
    '`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

    '''
    # Your code here
    top_articles = list(df.title.value_counts().index)[:n]

    return top_articles # Return the top article titles from df (not df_cont

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

    '''
    # Your code here
    top_articles = list(df.article_id.value_counts().index)[:n]
```

```
    return top_articles # Return the top article ids
```

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

```
['use deep learning for image classification', 'insights from new york car a
ccident reports', 'visualize car data with brunel', 'use xgboost, scikit-lea
rn & ibm watson machine learning apis', 'predicting churn with the spss rand
om tree algorithm', 'healthcare python streaming application demo', 'finding
optimal locations of new store using decision optimization', 'apache spark l
ab, part 1: basic concepts', 'analyze energy consumption in buildings', 'gos
ales transactions for logistic regression model']
['1429.0', '1330.0', '1431.0', '1427.0', '1364.0', '1314.0', '1293.0', '117
0.0', '1162.0', '1304.0']
```

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 wit
    an article and a 0 otherwise
    '''
    # Fill in the function here
    user_item = pd.pivot_table(df[['article_id', 'user_id']], index='user_ic
    user_item[user_item != 0] = 1

    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-a
assert user_item.shape[1] == 714, "Oops!  The number of articles in the user
assert user_item.sum(axis=1)[1] == 36, "Oops!  The number of articles seen b
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
                    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, excluding the sp
    similarity_df = user_item[user_item.index != user_id].dot(user_item[user

    # sort by similarity and convert to list
    most_similar_users = list(similarity_df.sort_values(user_id, ascending=F
```

```
    return most_similar_users # return a list of the users in order from mos
```

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

```
The 10 most similar users to user 1 are: [3933, 23, 3782, 203, 4459, 3870, 1
31, 46, 4201, 395]
The 5 most similar users to user 3933 are: [1, 23, 3782, 4459, 203]
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.head()
```

Out[ ]:

	article_id	title	email	user_id
0	1430.0	using pixiedust for fast, flexible, and easier...	ef5f11f77ba020cd36e1105a00ab868bbdbf7fe7	1
1	1314.0	healthcare python streaming application demo	083cbdfa93c8444beaa4c5f5e0f5f9198e4f9e0b	2
2	1429.0	use deep learning for image classification	b96a4f2e92d8572034b1e9b28f9ac673765cd074	3
3	1338.0	ml optimization using cognitive assistant	06485706b34a5c9bf2a0ecdac41daf7e7654ceb7	4
4	1276.0	deploy your python model as a restful api	f01220c46fc92c6e6b161b1849de11faacd7ccb2	5

In [ ]:
```
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
```

```
                (this is identified by the title column)
    '''
    # Your code here
    article_names = list(df[df.article_id.isin(article_ids)].title.unique())

    return article_names # Return the article names associated with list of


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
                    (this is identified by the doc_full_name column in df_co

    Description:
    Provides a list of the article_ids and article titles that have been see
    '''
    # Your code here
    user_df = user_item[user_item.index == user_id]
    article_ids = list(user_df.loc[:,(user_df == 1).any()].columns)

    article_names = get_article_names(article_ids, df=df)

    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
    Does this until m recommendations are found

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

    For the user where the number of recommended articles starts below m
    and ends exceeding m, the last items are chosen arbitrarily
    '''
    # Your code here

    #get similar users and ids of articles already seen
    similar_users = find_similar_users(user_id)
```

```
    seen_article_ids = get_user_articles(user_id)[0]

    #loop through other users
    recs = []
    for sim in similar_users:
        if len(recs) >= m:
            break
        else:
            sim_article_ids = get_user_articles(sim)[0]
            rec_article_ids = list(set(sim_article_ids)-set(seen_article_ids
            recs = recs + rec_article_ids
    recs = recs[:m]

    #recommend random articles if needed
    unseen_article_ids = list(set(df.article_id)-set(seen_article_ids))
    random.shuffle(unseen_article_ids)
    for article in unseen_article_ids:
        if len(recs) >= m:
            break
        else:
            recs = recs + article

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

In [ ]:
```
# Check Results
get_article_names(user_user_recs(1, 10)) # Return 10 recommendations for use
```

Out[ ]:
```
['visualize data with the matplotlib library',
 'analyze open data sets with pandas dataframes',
 'gosales transactions for logistic regression model',
 'simple graphing with ipython and\xa0pandas',
 '5 practical use cases of social network analytics: going beyond facebook
and twitter',
 'this week in data science (april 18, 2017)',
 'higher-order logistic regression for large datasets',
 'the power of machine learning in spark',
 'deep learning achievements over the past year ',
 'generalization in deep learning']
```

In [ ]:
```
# Test your functions here - No need to change this code - just run this cel
assert set(get_article_names(['1024.0', '1176.0', '1305.0', '1314.0', '1422.
assert set(get_article_names(['1320.0', '232.0', '844.0'])) == set(['housing
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
assert set(get_user_articles(2)[0]) == set(['1024.0', '1176.0', '1305.0', '1
assert set(get_user_articles(2)[1]) == set(['using deep learning to reconstr
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 t
                    num_interactions - the number of articles viewed by the

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

    '''
    # Your code here

    #build neighbors df
    neighbors_df = user_item[user_item.index != user_id].dot(user_item[user_
    neighbors_df['neighbor_id'] = neighbors_df.index
    neighbors_df.columns = ['similarity', 'neighbor_id']
    neighbors_df.index.name = None

    #add column for total interactions
    neighbor_item = user_item[user_item.index != user_id] #neighbor_item is
    neighbors_df['num_interactions'] = neighbor_item[neighbor_item.index ==

    #reorder cols
    neighbors_df = neighbors_df[['neighbor_id', 'similarity', 'num_interacti

    #sort and reset index
    neighbors_df = neighbors_df.sort_values(['similarity', 'num_interactions
    neighbors_df = neighbors_df.reset_index(drop=True)

    return neighbors_df # Return the dataframe specified in the doc_string


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 tit

    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
    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.

    '''
    # Your code here

    #get similar users and ids of articles already seen
    top_similar_users = get_top_sorted_users(user_id)
    seen_article_ids = get_user_articles(user_id)[0]

    #loop through other users
    recs = []
    for sim in top_similar_users:
        if len(recs) >= m:
            break
        else:
            sim_article_ids = get_user_articles(sim)[0]
            rec_article_ids = list(set(sim_article_ids)-set(seen_article_ids
            recs = recs + rec_article_ids
    recs = recs[:m]

    #recommend articles with most interactions if needed
    unseen_article_ids = list(set(get_top_article_ids((len(seen_article_ids)
    for article in unseen_article_ids:
        if len(recs) >= m:
            break
        else:
            recs.append(article)

    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 funct
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 name
print(rec_names)
```

```
The top 10 recommendations for user 20 are the following article ids:
['1330.0', '1304.0', '1314.0', '1436.0', '1398.0', '1162.0', '1293.0', '136
4.0', '1431.0', '1429.0']

The top 10 recommendations for user 20 are the following article names:
['healthcare python streaming application demo', 'use deep learning for imag
e classification', 'predicting churn with the spss random tree algorithm',
'analyze energy consumption in buildings', 'visualize car data with brunel',
'gosales transactions for logistic regression model', 'welcome to pixiedus
t', 'insights from new york car accident reports', 'finding optimal location
s of new store using decision optimization', 'total population by country']
```

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

user1_most_sim = get_top_sorted_users(1)['neighbor_id'][0] # Find the user t
user131_10th_sim = get_top_sorted_users(131)['neighbor_id'][10] # Find the 1
```

In [ ]:
```
## 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.

Provide your response here.

The **top_articles** function could be used to recommend articles to new users, as it does not rely on an existing user_id. A better method might be to compare other user data (e.g., country, industry, interests) between users and filter recommendations based on similar users.

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

# What would your recommendations be for this new user '0.0'?  As a new user
# Provide a list of the top 10 article ids you would give to
new_user_recs = get_top_article_ids(10) # Your recommendations here
```

In [ ]:
```
assert set(new_user_recs) == set(['1314.0','1429.0','1293.0','1427.0','1162.

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

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

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

Annnnnndddd SKIP!

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
user_id										
1	0.0	0.0	0.0	0.0	0.0	0.0	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
3	0.0	0.0	0.0	0.0	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
5	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.

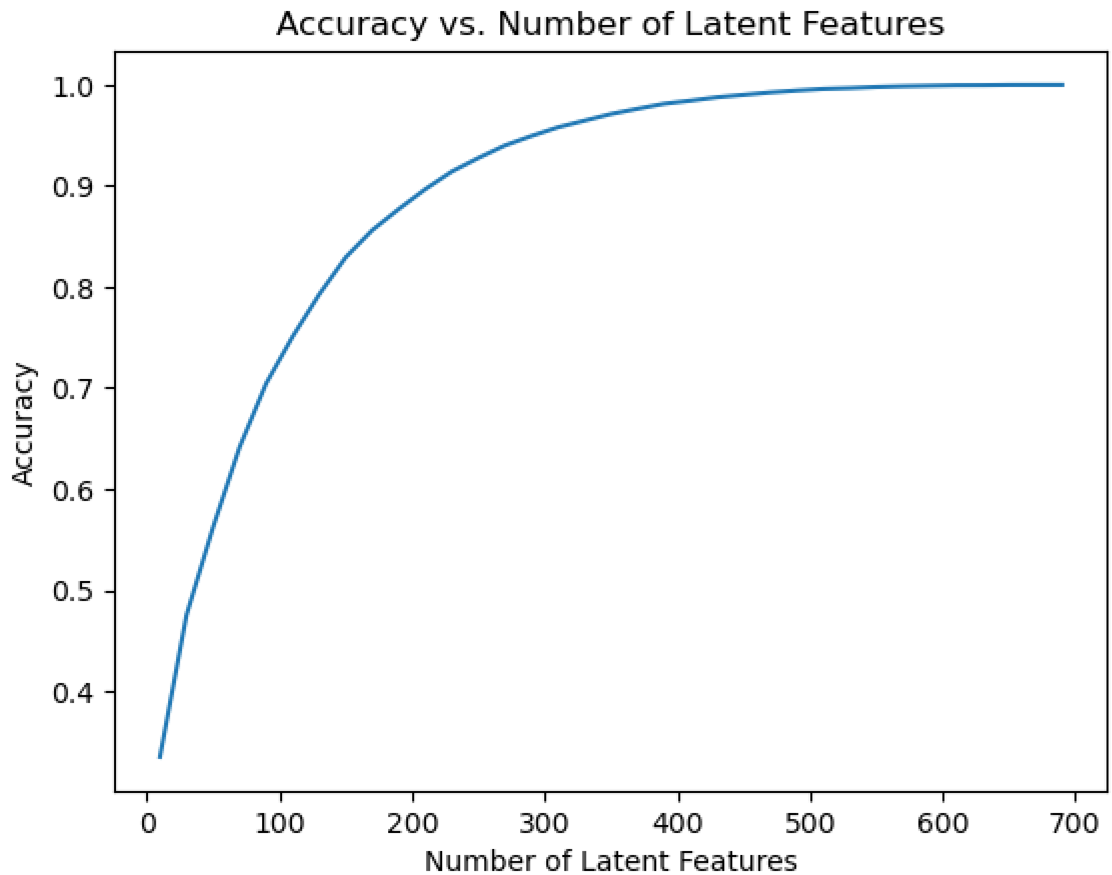
In [ ]:
```
# Perform SVD on the User-Item Matrix Here

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

Provide your response here.

This matrix simply gives 0 (no interaction) or 1 (user-articel interaction) rather than the ratings matrix used in the lesson. This method works because there are no missing values in the matrix.

`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 eac
    user_item_test - a user-item matrix of the testing dataframe
                    (unique users for each row and unique articles for each
    test_idx - all of the test user ids
    test_arts - all of the test article ids

    '''
    # Your code here
    user_item_train = create_user_item_matrix(df_train)
    user_item_test = create_user_item_matrix(df_test)
    test_idx = list(user_item_test.index)
    test_arts = list(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_trair
```

In [ ]:
```
user_item_test.shape
```

Out[ ]: (682, 574)

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
    '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 f
}

t.sol_4_test(sol_4_dict)
```

```
Awesome job!  That's right!  All of the test articles are in the training da
ta, but there are only 20 test users that were also in the training set.  Al
l of the other users that are in the test set we have no data on.  Therefor
e, we cannot 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
```

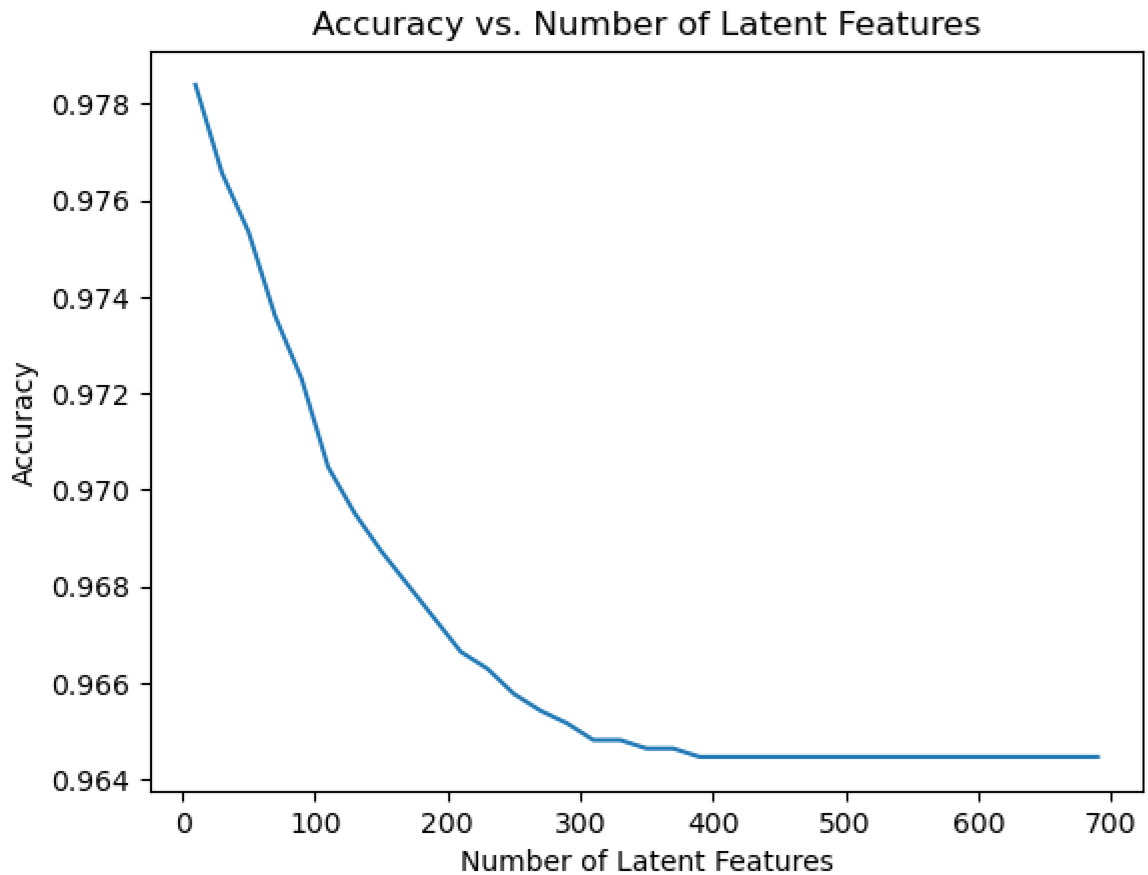
```
u_train, s_train, vt_train = np.linalg.svd(user_item_train) # fit svd simila
```

In [ ]:
```
#find subset of rows in user_item_test dataset that we can predict using the
user_intersect = np.intersect1d(user_item_train.index, test_idx)
subset = user_item_test.loc[user_intersect]

print('The following users are in both sets: {}'.format(list(subset.index)))
```

```
The following users are in both sets: [2917, 3024, 3093, 3193, 3527, 3532, 3
684, 3740, 3777, 3801, 3968, 3989, 3990, 3998, 4002, 4204, 4231, 4274, 4293,
4487]
```

In [ ]:
```
# Use these cells to see how well you can use the training
# decomposition to predict on test data
```

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

for k in num_latent_feats:
    # restructure with k latent features
    s_train_new, u_train_new, vt_train_new = np.diag(s_train[:k]), u_train[:

    #subset U and Vt matrices
    u_train_new_subset = u_train_new[user_item_train.index.isin(test_idx), :
    vt_train_new_subset = vt_train_new[:, user_item_train.columns.isin(test_

    # take dot product
    user_item_est = np.around(np.dot(np.dot(u_train_new_subset, s_train_new)

    # compute error for each prediction to actual value
    diffs = np.absolute(np.subtract(subset, 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)/(subset.shape[0]*subset.sh
plt.xlabel('Number of Latent Features');
plt.ylabel('Accuracy');
plt.title('Accuracy vs. Number of Latent Features');
```



`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?

Your response here.

Given that we can only make predictions for 20 users in these circumstances, splitting the data into train and test sets doesn't give us much to go on. Accuracy of predictions indeed goes *down* as the number of latent features increases.

To compare the SVD-based recommendations to the simple user-user collaborative recommendations from Part III, we could run an A/B test where the control group gets the "current" system from Part III and the experimental group gets the SVD-based recommendations. We could then compare the results of how many recommended articles the users in each group interact with.