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('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 [ ]:

```
# 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 [ ]:
```
# Calculate the median number of interactions by user
median_val = df['article_id'].groupby(df['email']).count().median()

# Calculate the maximum number of interactions by any user
max_views_by_user = df['article_id'].groupby(df['email']).count().max()
```

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

In [ ]:
```
# Find and explore duplicate articles
duplicate_articles = df_content.duplicated(subset=['article_id'], keep='first').sum
print(f"Number of duplicate articles: {duplicate_articles}")
```

```
Number of duplicate articles: 5
```

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

`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 [ ]:
```
# Number of unique articles that have at least one interaction
unique_articles = df['article_id'].nunique()

# Number of unique articles on the IBM platform
total_articles = df_content['article_id'].nunique()

# Number of unique users
unique_users = df['email'].nunique()

# Number of user-article interactions
user_article_interactions = df.shape[0]
```

`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 [ ]:
```
# The most viewed article in the dataset as a string with one value following the d
most_viewed_article_id = str(df['article_id'].value_counts().idxmax())

# The most viewed article in the dataset was viewed how many times?
max_views = df['article_id'].value_counts().max()
```

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

```
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 [ ]:
```
## 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 ______.`': us
    '`The maximum number of user-article interactions by any 1 user is ______.`': m
    '`The most viewed article in the dataset was viewed _____ times.`': max_views,
    '`The article_id of the most viewed article is ______.`': most_viewed_article_i
    '`The number of unique articles that have at least 1 rating ______.`': unique_a
    '`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

    '''
    # Get the top article ids, ensuring they are strings to match DataFrame convers
    top_article_ids = get_top_article_ids(n, df)
    # Filter df to only these top ids and ensure type consistency
    top_articles_df = df[df['article_id'].astype(str).isin(top_article_ids)].copy()
    top_articles_df['article_id'] = top_articles_df['article_id'].astype(str)  # Co
    # Create a ranking based on the order of top_article_ids and reorder DataFrame
    top_articles_df['rank'] = top_articles_df['article_id'].apply(lambda x: top_art
    top_articles_df.sort_values('rank', inplace=True)
    top_articles = top_articles_df['title'].drop_duplicates().tolist()

    return top_articles # 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

    '''
    # Count the number of interactions per article
    article_counts = df['article_id'].value_counts().head(n)
    top_article_ids = article_counts.index.astype(str).tolist()  # Ensure article I

    return top_article_ids # 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 accident
reports', 'visualize car data with brunel', 'use xgboost, scikit-learn & ibm watson
machine learning apis', 'predicting churn with the spss random tree algorithm', 'hea
lthcare python streaming application demo', 'finding optimal locations of new store
using decision optimization', 'apache spark lab, part 1: basic concepts', 'analyze e
nergy consumption in buildings', 'gosales transactions for logistic regression mode
l']
['1429.0', '1330.0', '1431.0', '1427.0', '1364.0', '1314.0', '1293.0', '1170.0', '11
62.0', '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.
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 val
    an article and a 0 otherwise
    '''
    # Create the user-article matrix with 1's and 0's
    # Step 1: Create a matrix with user ids as rows and article ids as columns
    # Set to 1 if a user has interacted with an article, regardless of how many tim
    user_item = df.groupby(['user_id', 'article_id']).size().unstack(fill_value=0)
    # Step 2: Convert any positive interactions to 1
```

```
    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-article
assert user_item.shape[1] == 714, "Oops!  The number of articles in the user-articl
assert user_item.sum(axis=1)[1] == 36, "Oops!  The number of articles seen by user
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 pro
                    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
    similarity = user_item.dot(user_item.loc[user_id])

    # Sort by similarity
    similarity = similarity.sort_values(ascending=False)

    # Create list of just the ids
    most_similar_users = similarity.index.tolist()

    # 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 le
```

In [ ]:
```
# Do a spot check of your function
print("The 10 most similar users to user 1 are: {}".format(find_similar_users(1)[:1
print("The 5 most similar users to user 3933 are: {}".format(find_similar_users(393
print("The 3 most similar users to user 46 are: {}".format(find_similar_users(46)[:
```

```
The 10 most similar users to user 1 are: [3933, 23, 3782, 203, 4459, 3870, 131, 420
1, 46, 5041]
The 5 most similar users to user 3933 are: [1, 23, 3782, 203, 4459]
The 3 most similar users to user 46 are: [4201, 3782, 23]
```

`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 [ ]:
```
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 arti
                    (this is identified by the title column)
    '''
    # Get the article names from df using the list of article_ids
    article_names = df[df['article_id'].astype(str).isin(article_ids)]['title'].dro

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


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 arti
                    (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
    '''
    # Find the articles that the user has interacted with
    user_row = user_item.loc[user_id]
    article_ids = user_row[user_row > 0].index.astype(str).tolist()
    article_names = get_article_names(article_ids, 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 them as
    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

    '''
    # Get a list of users similar to the provided user_id
    most_similar_users = find_similar_users(user_id, user_item)
    user_article_ids, _ = get_user_articles(user_id, user_item)

    # Find articles from similar users that the user hasn't seen
    recs = set()
    for similar_user in most_similar_users:
        sim_user_article_ids, _ = get_user_articles(similar_user, user_item)
        recs.update([aid for aid in sim_user_article_ids if aid not in user_article
        if len(recs) >= m:  # Ensure we don't exceed the number of required recomme
            break

    return list(recs)[:m]
```

In [ ]:

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

Out[ ]:

```
['ml optimization using cognitive assistant',
 'graph-based machine learning',
 'optimizing a marketing campaign: moving from predictions to actions',
 'movie recommender system with spark machine learning',
 'car performance data',
 'easy json loading and social sharing in dsx notebooks',
 'working with db2 warehouse on cloud in data science experience',
 'this week in data science (may 30, 2017)',
 'higher-order logistic regression for large datasets',
 'ml algorithm != learning machine']
```

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', '14
assert set(get_article_names(['1320.0', '232.0', '844.0'])) == set(['housing (2015)
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 demogra
assert set(get_user_articles(2)[0]) == set(['1024.0', '1176.0', '1305.0', '1314.0',
assert set(get_user_articles(2)[1]) == set(['using deep learning to reconstruct hig
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 prov
                    num_interactions - the number of articles viewed by the user -

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

    '''
    # Calculate similarity
    similarity = user_item.dot(user_item.loc[user_id])
    similarity = similarity.drop(user_id).reset_index()
    similarity.columns = ['neighbor_id', 'similarity']

    # Calculate number of interactions
    num_interactions = df.groupby('user_id')['article_id'].count()

    # Combine the data
    neighbors_df = similarity
    neighbors_df['num_interactions'] = neighbors_df['neighbor_id'].apply(lambda x:

    # Sort by similarity and then by number of interactions
    neighbors_df = neighbors_df.sort_values(by=['similarity', 'num_interactions'],

    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 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
    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.

    '''
    # Ensure the article IDs are of a consistent type
    df['article_id'] = df['article_id'].astype(str)

    # Get sorted users
    neighbors_df = get_top_sorted_users(user_id, df, user_item)

    # Articles already seen by the user
    seen_articles = set(user_item.loc[user_id][user_item.loc[user_id] == 1].index.a

    # Grouping articles by interaction count, ensuring article_id is treated as str
    article_interactions = df.groupby('article_id').count()['user_id']

    # Initialize recommendations
    recs = []

    for neighbor in neighbors_df['neighbor_id']:
        # Get articles viewed by the neighbor, ensuring IDs are strings
        neighbor_articles = set(user_item.loc[neighbor][user_item.loc[neighbor] ==
        # Articles not seen by user
        new_recs = list(neighbor_articles - seen_articles)
        # Filter new_recs and sort by the number of interactions
        if new_recs:
            recs_to_add = article_interactions.loc[new_recs].sort_values(ascending=
            recs.extend(recs_to_add)
        if len(recs) >= m:
            break

    recs = recs[:m]
    # Get article names
    rec_names = list(df[df['article_id'].isin(recs)]['title'].unique())

    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:
['1330.0', '1427.0', '1364.0', '1170.0', '1162.0', '1304.0', '1351.0', '1160.0', '13
54.0', '1368.0']

The top 10 recommendations for user 20 are the following article names:
['apache spark lab, part 1: basic concepts', 'predicting churn with the spss random
tree algorithm', 'analyze energy consumption in buildings', 'use xgboost, scikit-lea
rn & ibm watson machine learning apis', 'putting a human face on machine learning',
'gosales transactions for logistic regression model', 'insights from new york car ac
cident reports', 'model bike sharing data with spss', 'analyze accident reports on a
mazon emr spark', 'movie recommender system with spark machine learning']
```

`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).iloc[0]['neighbor_id']  # Most similar to
user131_10th_sim = get_top_sorted_users(131).iloc[9]['neighbor_id']  # 10th most si

print("The most similar user to user 1 is:", user1_most_sim)
print("The 10th most similar user to user 131 is:", user131_10th_sim)
```

```
The most similar user to user 1 is: 3933
The 10th most similar user to user 131 is: 242
```

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.

Existing Functions: For a new user, also known as the cold-start problem in recommender systems, collaborative filtering (user-user recommendations) isn't immediately useful because there's no history of interactions to compare with other users. Therefore, we can't use `user_user_recs` or `user_user_recs_part2` directly.

A Better Way for New Users:

1. **Popularity-Based Recommendations**: One straightforward approach is to recommend the most popular articles across the platform. These can be obtained using the `get_top_article_ids` function, which will provide articles that have the highest number of interactions, assuming these might be universally appealing.

2. **Content-Based Filtering**: If any demographic data or content preferences are available when the user signs up (e.g., through a signup questionnaire), content-based filtering could be utilized to recommend articles similar to the user's indicated preferences.

3. **Hybrid Methods**: Combining content-based filtering with popularity metrics or using machine learning to predict user preferences based on limited initial inputs (like session duration on articles, etc.).

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

In [ ]:
```
assert set(new_user_recs) == set(['1314.0','1429.0','1293.0','1427.0','1162.0','136

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
```

```python
user_item_matrix = pd.read_pickle('user_item_matrix.p')
```

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

Out[ ]:

article_id user_id	0.0	100.0	1000.0	1004.0	1006.0	1008.0	101.0	1014.0	1015.0	1016.0	...
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 [ ]:
```python
# Perform SVD on the User-Item Matrix Here

u, s, vt = np.linalg.svd(user_item_matrix, full_matrices=False) # use the built in

# Display shapes of the matrices
print("Shape of U:", u.shape)
print("Shape of S:", s.shape)
print("Shape of VT:", vt.shape)
```

```
Shape of U: (5149, 714)
Shape of S: (714,)
Shape of VT: (714, 714)
```

SVD is used here differently from many typical recommendation system lessons, particularly those dealing with movie ratings. Here's why:

1. **Binary Data vs. Ratings**:
   - **In Lessons**: SVD often handles matrices with ratings, where each entry is a user's rating for a product, often on a scale (like 1-5 stars).
   - **Here**: The matrix is binary (1s and 0s), indicating whether a user has interacted with an article. There are no ratings to indicate the strength or preference of the interaction.
2. **Missing Data Handling**:
   - **In Lessons**: SVD cannot be directly applied when there are missing data (NaN values) because it requires a complete matrix. Techniques like collaborative filtering

might use matrix factorization that handles missing values implicitly (e.g., using stochastic gradient descent or ALS).
   - **Here**: The matrix is fully populated with 0s (no interaction) and 1s (interaction). This allows the direct application of classical SVD without needing to impute or handle missing data explicitly.
3. **Purpose and Outcome**:
   - **In Lessons**: SVD is typically used to predict ratings and uncover latent factors that explain observed ratings.
   - **Here**: SVD is used to uncover latent patterns in user-article interactions, such as underlying topics or types of articles that users tend to interact with, even if they haven't rated them.

Practical Implications

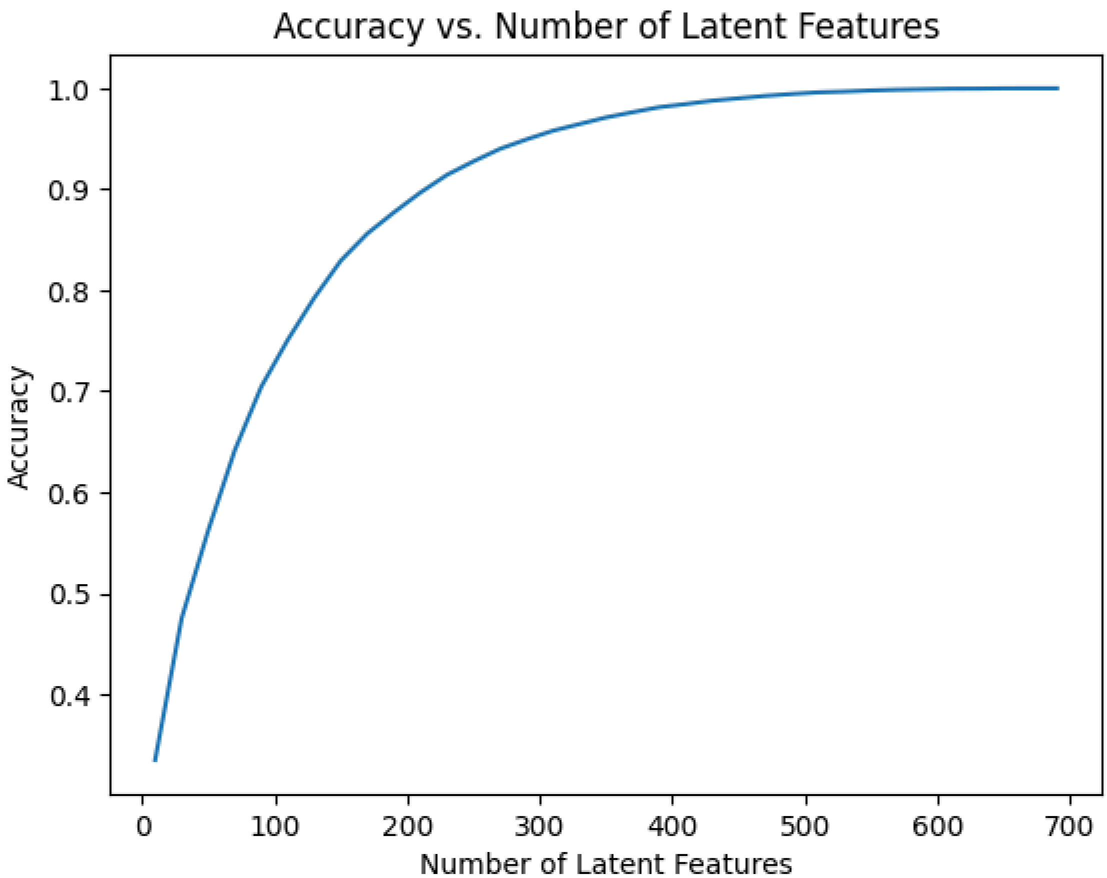
Using SVD in this binary interaction scenario allows us to identify dimensions that capture the most significant patterns of interaction across articles, potentially leading to better personalized recommendations. Users similar in lower-dimensional latent space can be considered similar based on the types of articles they interact with, not just specific articles they've both seen.

`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 [ ]:
```python
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), axis=None), axis=None)
    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');
```

```
c:\Users\kilgo\AppData\Local\Programs\Python\Python311\Lib\site-packages\numpy\core
\fromnumeric.py:86: FutureWarning: The behavior of DataFrame.sum with axis=None is d
eprecated, in a future version this will reduce over both axes and return a scalar.
To retain the old behavior, pass axis=0 (or do not pass axis)
  return reduction(axis=axis, out=out, **passkwargs)
```



`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 colum
    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 user-item matrices for the training and test sets
    user_item_train = df_train.groupby(['user_id', 'article_id']).size().unstack(fi
    user_item_test = df_test.groupby(['user_id', 'article_id']).size().unstack(fill

    # Identify unique user IDs and article IDs in the test set
    test_idx = user_item_test.index.tolist()  # list of user ids in the test set
    test_arts = user_item_test.columns.tolist()  # list of article ids in the test

    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_i
```

In [ ]:

```
# Set of user and article IDs in the training data
train_idx = set(user_item_train.index)
train_arts = set(user_item_train.columns)

# Set of user and article IDs in the test data
test_idx_set = set(test_idx)
test_arts_set = set(test_arts)

# Users and articles in the test set that can be predicted (i.e., also exist in the
predictable_users = test_idx_set.intersection(train_idx)
predictable_articles = test_arts_set.intersection(train_arts)

# Cold start problem: users and articles in the test set not found in the training
cold_start_users = test_idx_set - train_idx
cold_start_articles = test_arts_set - train_arts

# Replace the values in the dictionary below
a = len(predictable_users)  # How many users can we make predictions for in the tes
b = len(cold_start_users)  # How many users in the test set are we not able to make
c = len(predictable_articles)  # How many articles can we make predictions for in t
d = len(cold_start_articles)  # How many articles in the test set are we not able t



sol_4_dict = {
    'How many users can we make predictions for in the test set?': a,
    'How many users in the test set are we not able to make predictions for because
    'How many articles can we make predictions for in the test set?': c,
```

```
    'How many articles in the test set are we not able to make predictions for beca
}

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 were also in the training set.  All of the other u
sers that are in the test set we have no data on.  Therefore, we cannot make predict
ions 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, full_matrices=False) #

# Display shapes of the decomposed matrices
print("Shape of U_train:", u_train.shape)
print("Shape of S_train:", s_train.shape)
print("Shape of VT_train:", vt_train.shape)
```

```
Shape of U_train: (4487, 714)
Shape of S_train: (714,)
Shape of VT_train: (714, 714)
```

In [ ]:
```
# Identifying users in both train and test sets (overlap)
test_users = user_item_test.index.intersection(user_item_train.index)

# Subsetting the U matrix to include only users in both train and test sets
u_test = u_train[user_item_train.index.isin(test_users), :]

# Subsetting the Vt matrix to include only articles in both train and test sets
vt_test = vt_train[:, user_item_train.columns.isin(test_arts)]

# User-item matrix for the test set (for users we can predict)
user_item_test_subset = user_item_test.loc[test_users]

print(user_item_test_subset.shape)
```

```
(20, 574)
```

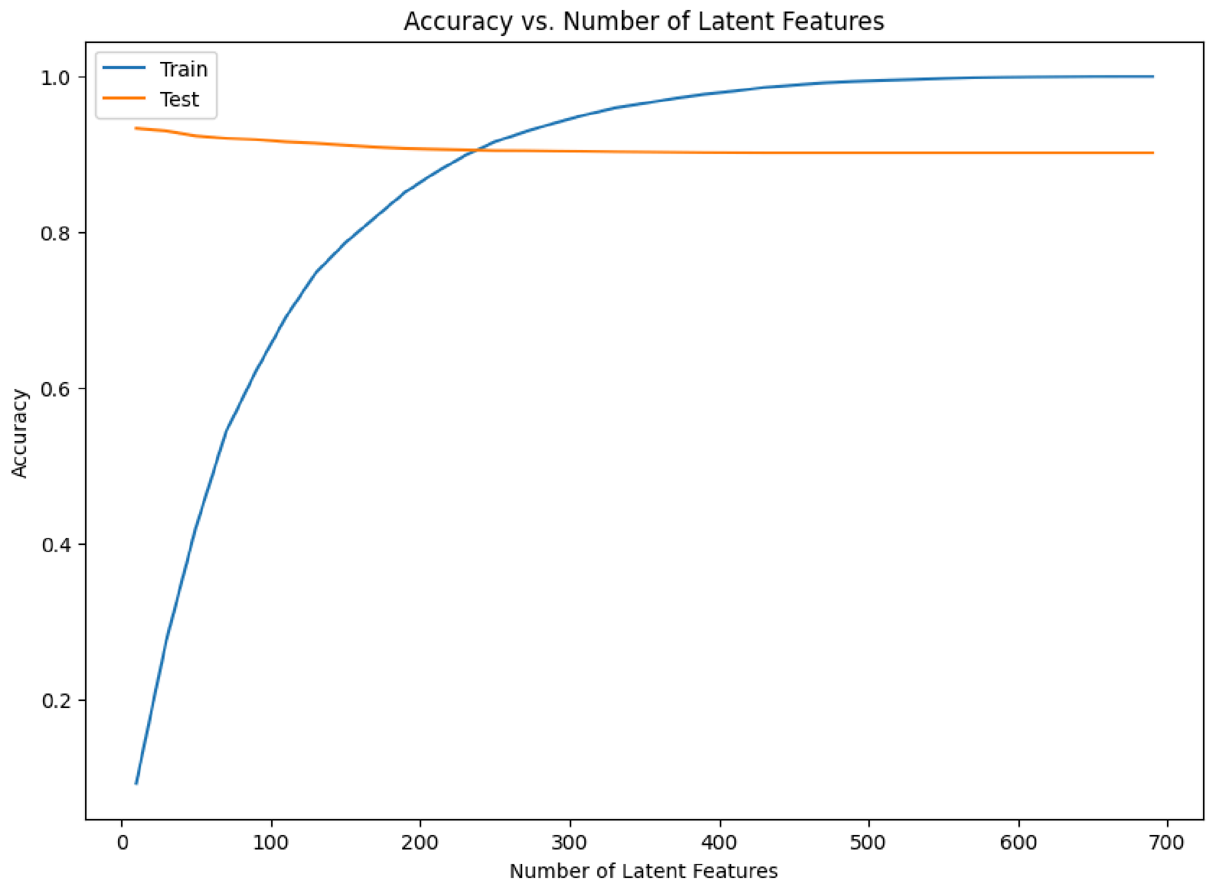
In [ ]:
```
num_latent_feats = np.arange(10,700+10,20)
sum_train_errs = []
sum_test_errs = []

for k in num_latent_feats:
    # restructure with k latent features
    s_train_k, u_train_k, vt_train_k = np.diag(s_train[:k]), u_train[:, :k], vt_tra
    u_test_k, vt_test_k = u_test[:, :k], vt_test[:k, :]
```

```
    # take dot product
    user_item_train_preds = np.around(np.dot(np.dot(u_train_k, s_train_k), vt_train
    user_item_test_preds = np.around(np.dot(np.dot(u_test_k, s_train_k), vt_test_k)

    # compute error for each prediction to actual value
    diffs_train = np.subtract(user_item_train, user_item_train_preds)
    diffs_test = np.subtract(user_item_test_subset, user_item_test_preds)

    # total errors and keep track of them
    train_err = np.sum(np.sum(np.abs(diffs_train), axis=None), axis=None)
    test_err = np.sum(np.sum(np.abs(diffs_test), axis=None), axis=None)

    sum_train_errs.append(train_err)
    sum_test_errs.append(test_err)
```

```
c:\Users\kilgo\AppData\Local\Programs\Python\Python311\Lib\site-packages\numpy\core
\fromnumeric.py:86: FutureWarning: The behavior of DataFrame.sum with axis=None is d
eprecated, in a future version this will reduce over both axes and return a scalar.
To retain the old behavior, pass axis=0 (or do not pass axis)
  return reduction(axis=axis, out=out, **passkwargs)
```

In [ ]:

```
plt.figure(figsize=(10, 7))
plt.plot(num_latent_feats, 1 - np.array(sum_train_errs)/df_train.shape[0], label='T
plt.plot(num_latent_feats, 1 - np.array(sum_test_errs)/df_test.shape[0], label='Tes
plt.xlabel('Number of Latent Features')
plt.ylabel('Accuracy')
plt.title('Accuracy vs. Number of Latent Features')
plt.legend()
plt.show()
```



`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?

The code above calculates the prediction accuracy for both the training and testing data as we vary the number of latent features. Observations could include:

- **Overfitting**: As the number of latent features increases, the model might fit the training data increasingly well, but the accuracy on the test data might start to decline. This suggests overfitting, where the model learns to perfectly predict the training data but performs poorly on unseen data.
- **Optimal Latent Features**: The point where the test accuracy starts to decline indicates the optimal number of latent features that balances between underfitting and overfitting.
- **Improvement Strategy**: To determine if these recommendations improve how users find articles, you could implement A/B testing to compare the behavior of users who receive these SVD-based recommendations versus those who receive random or popularity-based recommendations. Metrics such as increased interaction rates or higher satisfaction scores could indicate success.

This analysis and the subsequent implementation of a test plan will help refine the recommendation system to ensure it effectively enhances user engagement.

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!