```
#%% md
# Recommendation System Project: IBM Community

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](https://review.udacity.com/#!/rubrics/3325/view).  **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

I. [Exploratory Data Analysis](#Exploratory-Data-Analysis)<br>
II. [Rank Based Recommendations](#Rank)<br>
III. [User-User Based Collaborative Filtering](#User-User)<br>
IV. [Content Based Recommendations](#Content-Recs)<br>
V. [Matrix Factorization](#Matrix-Fact)<br>
VI. [Extras & Concluding](#conclusions)

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.
#%%
import pandas
# libraries
import pandas as pd
```

```
26 import numpy as np
27 import matplotlib.pyplot as plt
28 import project_tests as t
29 import seaborn as sns
30
31 #%%
32 # load data
33 df = pd.read_csv(
34     'data/user-item-interactions.csv',
35     dtype={'article_id': int, 'title': str, 'email'
   : str}
36 )
37 # Show df to get an idea of the data
38 df.head()
39 #%% md
40 ### <a class="anchor" id="Exploratory-Data-Analysis
   ">Part I : Exploratory Data Analysis</a>
41
42 Use the dictionary and cells below to provide some
   insight into the descriptive statistics of the data
   .
43
44 `1.` Are there any missing values? If so, provide a
    count of missing values. If there are missing
   values in `email`, assign it the same id value `"
   unknown_user"`.
45 #%%
46 # Some interactions do not have a user associated
   with it, assume the same user.
47 df.info()
48 #%%
49 print(f"Number of Null email values is: ")
50 #%%
51 df[df.email.isna()]
52 #%%
53 # Fill email NaNs with "unknown_user"
54 df.loc[df['email'].isna(), 'email'] = 'unknown_user
   '
55 #%%
56 # Check if no more NaNs
57 df[df.email.isna()]
```

```
#%% md
`2.` 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.
#%%
# What are the descriptive statistics of the number of articles a user interacts with?
df_user_articles = df.groupby('email')['article_id'].count().reset_index()
df_user_articles.columns = ['email', 'article_count']
df_user_articles["article_count"].describe()
#%%
# Create a plot of the number of articles read by each user
sns.barplot(data=df_user_articles, x='article_count', y='email')
plt.xlabel('number of articles')
plt.ylabel('number of users')
plt.yticks([])
plt.title('Number of Users Reading Articles')
plt.show()
#%%
article_counts = df.groupby('article_id')['email'].count().reset_index()
article_counts.columns = ['article_id', 'read_count']
article_counts['read_count'].describe()
#%%
# Create a plot of the number of times each article was read

sns.barplot(data=article_counts, y='read_count', x='article_id')
plt.xlabel('number of users')
plt.ylabel('number of articles')
plt.xticks([])
plt.title('Distribution of Article Usage')
plt.show()
```

```
#%%
# Fill in the median and maximum number of
   user_article interactions below

median_val = np.median(df_user_articles["
   article_count"]) # 50% of individuals interact
   with 3 number of articles or fewer.
max_views_by_user =  np.max(df_user_articles["
   article_count"])# The maximum number of user-
   article interactions by any 1 user is 364.
#%% md
`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
   ).<br>
**c.** The number of unique users in the dataset
   . (excluding null values) <br>
**d.** The number of user-article interactions in
   the dataset.
#%%
unique_articles = len(article_counts[
   article_counts["read_count"] >= 1])
 # The number of unique articles that have at
   least one interaction
total_articles = len(df["article_id"].unique()) #
   The number of unique articles on the IBM platform
unique_users =  len(df["email"].unique()) # The
   number of unique users
user_article_interactions = sum(article_counts["
   read_count"]) # The number of user-article
   interactions
#%% md
`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
```

```
105 of these null values likely belonged to a single
    user (which is how they are stored using the
    function below).
106 #%%
107 most_viewed_article_id =  article_counts.loc[
    article_counts["read_count"].idxmax(), "article_id
    "]# The most viewed article in the dataset as a
    string with one value following the decimal
108 max_views =  article_counts["read_count"].max()#
    The most viewed article in the dataset was viewed
    how many times?
109 #%%
110 ## No need to change the code here - this will be
    helpful for later parts of the notebook
111 # Run this cell to map the user email to a user_id
     column and remove the email column
112
113 def email_mapper(df=df):
114     coded_dict = {
115         email: num
116         for num, email in enumerate(df['email'].
    unique(), start=1)
117     }
118     return [coded_dict[val] for val in df['email'
    ]]
119
120 df['user_id'] = email_mapper(df)
121 del df['email']
122
123 # show header
124 df.head()
125 #%%
126 ## If you stored all your results in the variable
    names above,
127 ## you shouldn't need to change anything in this
    cell
128
129 sol_1_dict = {
130     ``50% of individuals have _____ or fewer
    interactions.``: median_val,
131     ``The total number of user-article
```

```
131 interactions in the dataset is ______.``:
    user_article_interactions,
132     ``The maximum number of user-article
    interactions by any 1 user is ______.``:
    max_views_by_user,
133     ``The most viewed article in the dataset was
    viewed _____ times.``: max_views,
134     ``The article_id of the most viewed article is
     ______.``: most_viewed_article_id,
135     ``The number of unique articles that have at
    least 1 rating ______.``: unique_articles,
136     ``The number of unique users in the dataset is
     ______``: unique_users,
137     ``The number of unique articles on the IBM
    platform``: total_articles
138 }
139
140 # Test your dictionary against the solution
141 t.sol_1_test(sol_1_dict)
142 #%% md
143 ### <a class="anchor" id="Rank">Part II: Rank-
    Based Recommendations</a>
144
145 In this project, 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.
146
147 `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.
148 #%%
149 def get_top_articles(n, df=df):
150     """
151     INPUT:
152     n - (int) the number of top articles to return
153     df - (pandas dataframe) df as defined at the
    top of the notebook
154
```

```
155     OUTPUT:
156     top_articles - (list) A list of the top 'n'
    article titles
157
158     """
159     top_articles = df.groupby('article_id')['
    user_id'].count().reset_index()
160     top_articles = top_articles.sort_values(by="
    user_id", ascending= False)
161     top_articles = top_articles["article_id"].iloc
    [0:n]
162     top_articles = df[df["article_id"].isin(
    top_articles)]["title"].unique().tolist()
163
164     return top_articles # Return the top article
    titles from df
165
166 def get_top_article_ids(n, df=df):
167     """
168     INPUT:
169     n - (int) the number of top articles to return
170     df - (pandas dataframe) df as defined at the
    top of the notebook
171
172     OUTPUT:
173     top_articles - (list) A list of the top 'n'
    article ids
174
175     """
176     top_articles = df.groupby('article_id')['
    user_id'].count().reset_index()
177     top_articles = top_articles.sort_values(by="
    user_id", ascending= False)
178     top_articles = top_articles["article_id"].iloc
    [0:n].tolist()
179
180     return top_articles # Return the top article
    ids
181 #%%
182 print(get_top_articles(10))
183 print(get_top_article_ids(10))
```

```
184 #%%
185 # Test your function by returning the top 5, 10,
    and 20 articles
186 top_5 = get_top_articles(5)
187 top_10 = get_top_articles(10)
188 top_20 = get_top_articles(20)
189
190 # Test each of your three lists from above
191 t.sol_2_test(get_top_articles)
192 #%% md
193 ### <a class="anchor" id="User-User">Part III:
    User-User Based Collaborative Filtering</a>
194
195
196 `1.` Use the function below to reformat the **df
    ** dataframe to be shaped with users as the rows
    and articles as the columns.
197
198 * Each **user** should only appear in each **row
    ** once.
199
200
201 * Each **article** should only show up in one **
    column**.
202
203
204 * **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.
205
206
207 * **If a user has not interacted with an item,
    then place a zero where the user-row meets for
    that article-column**.
208
209 Use the tests to make sure the basic structure of
    your matrix matches what is expected by the
    solution.
```

```
210 #%%
211 # create the user-article matrix with 1's and 0's
212
213 def create_user_item_matrix(df, fill_value=0):
214     """
215     INPUT:
216     df - pandas dataframe with article_id, title,
    user_id columns
217
218     OUTPUT:
219     user_item - user item matrix
220
221     Description:
222     Return a matrix with user ids as rows and
    article ids on the columns with 1 values where a
    user interacted with
223     an article and a 0 otherwise
224     """
225
226     # convert to string to ensure correct column
    renaming
227
228     data = df.copy()
229     data['article_id'] = data['article_id'].astype
    (str)
230
231     unique_article_ids = data['article_id'].unique
    ()
232
233     # pivot data
234     user_item = data[data['article_id'].isin(
    unique_article_ids)].pivot_table(
235         index='user_id',
236         columns='article_id',
237         aggfunc='size',  # Count occurrences of
    interactions
238         fill_value=0
239     )
240
241     # ensure that pivoted table is binary
242     user_item[user_item >= 1] = 1
```

```
243
244     return user_item # return the user_item matrix

245
246 user_item = create_user_item_matrix(df)
247 #%%
248 ## Tests: You should just need to run this cell.
    Don't change the code.
249 assert user_item.shape[0] == 5149, "Oops!  The
    number of users in the user-article matrix doesn't
     look right."
250 assert user_item.shape[1] == 714, "Oops!  The
    number of articles in the user-article matrix
    doesn't look right."
251 assert user_item.sum(axis=1)[1] == 36, "Oops!  The
     number of articles seen by user 1 doesn't look
    right."
252 print("You have passed our quick tests!  Please
    proceed!")
253 #%% md
254 `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.
255
256 Use the tests to test your function.
257 #%%
258 # Lets use the cosine_similarity function from
    sklearn
259 from sklearn.metrics.pairwise import
    cosine_similarity
260 #%%
261 def find_similar_users(user_id, user_item=
    user_item, include_similarity=False):
262     """
263     INPUT:
```

```
264     user_id - (int) a user_id
265     user_item - (pandas dataframe) matrix of users
     by articles:
266                 1's when a user has interacted
    with an article, 0 otherwise
267     include_similarity - (bool) whether to include
     the similarity in the output
268
269     OUTPUT:
270     similar_users - (list) an ordered list where
    the closest users (largest dot product users)
271                     are listed first
272
273     Description:
274     Computes the similarity of every pair of users
     based on the dot product
275     Returns an ordered list of user ids. If
    include_similarity is True, returns a list of
    lists
276     where the first element is the user id and the
     second the similarity.
277
278     """
279
280     # initialize user to be tested
281     user_id -=1 # correct for 0 indexing in python

282     user1 = user_item.iloc[user_id].values.reshape
    (1, -1)
283     row_ids = range(len(user_item))
284     user_ids = range(1, len(user_item)+1) #
    user_ids start with 1
285     similarity = []
286
287     # iterate across user rows and calculate
    cosine similarity
288     for row in row_ids:
289         user2 = user_item.iloc[row].values.reshape
    (1, -1)
290         sim = cosine_similarity(user1, user2)
291         similarity.append(sim[0][0])
```

```
292
293     # store similarity next to user_id
294     sim_df = pd.DataFrame({"similarity":
    similarity,
295                            "user_id": user_ids})
296     # remove the own user's id
297     sim_df = sim_df.drop(user_id)
298
299     # sort by similarity
300     sim_df = sim_df.sort_values(by=["similarity"
    ], ascending= False)
301
302     # create list of just the ids
303     ids = sim_df["user_id"].tolist()
304     # create list of just the similarities
305     sim = sim_df["similarity"].tolist()
306
307     if include_similarity:
308         return  ids, sim
309     return ids # return a list of the users in
    order from most to least similar
310
311 #%%
312 # Do a spot check of your function
313 print("The 10 most similar users to user 1 are
    : {}".format(find_similar_users(1)[:10]))
314 print("The 5 most similar users to user 3933 are
    : {}".format(find_similar_users(3933)[:5]))
315 print("The 3 most similar users to user 46 are
    : {}".format(find_similar_users(46)[:3]))
316 #%% md
317 `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.
318 #%%
319 def get_article_names(article_ids, df=df):
320     """
321     INPUT:
322     article_ids - (list) a list of article ids
```

```
323     df - (pandas dataframe) df as defined at the
    top of the notebook
324
325     OUTPUT:
326     article_names - (list) a list of article names
     associated with the list of article ids
327                         (this is identified by the
    title column in df)
328     """
329
330     article_names = df.loc[df['article_id'].isin(
    article_ids), "title"].unique()
331
332
333     return article_names # Return the article
    names associated with list of article ids
334
335 def get_ranked_article_unique_counts(article_ids,
    user_item=user_item):
336     """
337     INPUT:
338     user_id - (int) a user id
339     user_item - (pandas dataframe) matrix of users
     by articles:
340                     1's when a user has interacted
    with an article, 0 otherwise
341
342     OUTPUT:
343     article_counts - (list) a list of tuples with
    article_id and number of
344                          unique users that have
    interacted with the article, sorted
345                          by the number of unique users
     in descending order
346
347     Description:
348     Provides a list of the article_ids and the
    number of unique users that have
349     interacted with the article using the
    user_item matrix, sorted by the number
350     of unique users in descending order
```

```
351     """
352     article_ids_str = list(map(str, article_ids))
353     rauc = pd.DataFrame({"article_id": article_ids
    ,
354                          "interactions": user_item[
    article_ids_str].sum()})
355     rauc = rauc.sort_values(by="interactions",
    ascending= False)
356
357     ranked_article_unique_counts = rauc[["
    article_id", "interactions"]].values.tolist()
358
359     return ranked_article_unique_counts
360
361
362 def get_user_articles(user_id, user_item=user_item
    ):
363     """
364     INPUT:
365     user_id - (int) a user id
366     user_item - (pandas dataframe) matrix of users
     by articles:
367                   1's when a user has interacted
    with an article, 0 otherwise
368
369     OUTPUT:
370     article_ids - (list) a list of the article ids
     seen by the user
371     article_names - (list) a list of article names
     associated with the list of article ids
372                         (this is identified by the
    title column in df)
373
374     Description:
375     Provides a list of the article_ids and article
     titles that have been seen by a user
376     """
377
378     user = user_item.iloc[user_id-1]
379
380     article_ids = user[user == 1].index.tolist()
```

```
381
382     article_ids = list(map(int, article_ids))
383
384     article_names = get_article_names(article_ids)
385
386
387     return article_ids, article_names # return the
     ids and names
388
389
390 def user_user_recs(user_id, m=10):
391     """
392     INPUT:
393     user_id - (int) a user id
394     m - (int) the number of recommendations you
    want for the user
395
396     OUTPUT:
397     recs - (list) a list of recommendations for
    the user
398
399     Description:
400     Loops through the users based on closeness to
    the input user_id
401     For each user - finds articles the user hasn't
     seen before and provides them as recs
402     Does this until m recommendations are found
403
404     Notes:
405     Users who are the same closeness are chosen
    arbitrarily as the 'next' user
406
407     For the user where the number of recommended
    articles starts below m
408     and ends exceeding m, the last items are
    chosen arbitrarily
409
410     """
411
412     similar_users, similar_score =
    find_similar_users(user_id, include_similarity=
```

```
412 True)
413     recs = [] # initalize recs
414     user = user_item.iloc[user_id] # initialize
    user
415
416     for u in range(0,len(similar_users)):
417         id = similar_users[u]
418         sim_user = user_item.iloc[id]
419         result = user_item.columns[(sim_user == 1
    ) & (user == 0)].tolist()
420
421         recs.extend([item for item in result if
    item not in recs])
422
423         # Check if we have enough recommendations
424         n_recs = len(recs)
425         if n_recs >= m:
426             break  # Exit loop if we have enough
    recommendations
427
428         recs = recs[:m]
429
430     recs = list(map(int, recs))
431
432     return recs # return your recommendations for
    this user_id
433 #%%
434 # Check Results
435 get_article_names(user_user_recs(1, 10)) # Return
    10 recommendations for user 1
436 #%%
437 get_ranked_article_unique_counts([1320, 232, 844])
438 #%%
439 get_article_names([1024, 1176, 1305, 1314, 1422,
    1427])
440 #%%
441 # Test your functions here - No need to change
    this code - just run this cell
442 assert set(get_article_names([1024, 1176, 1305,
    1314, 1422, 1427])) == set(['using deep learning
    to reconstruct high-resolution audio', 'build a
```

```
python app on the streaming analytics service', '
    gosales transactions for naive bayes model', '
    healthcare python streaming application demo', '
    use r dataframes & ibm watson natural language
    understanding', 'use xgboost, scikit-learn & ibm
    watson machine learning apis']), "Oops! Your the
    get_article_names function doesn't work quite how
    we expect."
assert set(get_article_names([1320, 232, 844
    ])) == set(['housing (2015): united states
    demographic measures','self-service data
    preparation with ibm data refinery','use the
    cloudant-spark connector in python notebook']), "
    Oops! Your the get_article_names function doesn't
    work quite how we expect."
assert set(get_user_articles(20)[0]) == set([1320
    , 232, 844])
assert set(get_user_articles(20)[1]) == set(['
    housing (2015): united states demographic measures
    ', 'self-service data preparation with ibm data
    refinery','use the cloudant-spark connector in
    python notebook'])
assert set(get_user_articles(2)[0]) == set([1024,
    1176, 1305, 1314, 1422, 1427])
assert set(get_user_articles(2)[1]) == set(['using
     deep learning to reconstruct high-resolution
    audio', 'build a python app on the streaming
    analytics service', 'gosales transactions for
    naive bayes model', 'healthcare python streaming
    application demo', 'use r dataframes & ibm watson
    natural language understanding', 'use xgboost,
    scikit-learn & ibm watson machine learning apis'])
assert get_ranked_article_unique_counts([1320, 232
    , 844])[0] == [1320, 123], "Oops! Your the
    get_ranked_article_unique_counts function doesn't
    work quite how we expect.\nMake sure you are using
     the user_item matrix to create the article counts
    ."
print("If this is all you see, you passed all of
    our tests!  Nice job!")
#%% md
```

```
451 `4.` Now we are going to improve the consistency
    of the **user_user_recs** function from above.
452
453 * 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.
454
455
456 * 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.
457 #%%
458 user_id = 2
459 user_ids, sim = find_similar_users(user_id,
    include_similarity=True)
460
461 neighbors_df = pd.DataFrame({'neighbor_id':
    user_ids,
462                                          'similarity':
    sim})
463
464
465 #%%
466 num_interactions = []
467 for row in range(len(neighbors_df)):
468     id = int(neighbors_df.iloc[row]['neighbor_id'
    ])-1 # correct id for 0 start
469     num_interactions.append(user_item.iloc[id].sum
    ())
470
471 neighbors_df["num_interactions"] =
    num_interactions
472 #%%
473 def get_top_sorted_users(user_id, user_item=
```

```
473 user_item):
474     """
475     INPUT:
476     user_id - (int)
477     user_item - (pandas dataframe) matrix of users
     by articles:
478             1's when a user has interacted with an
     article, 0 otherwise
479 
480 
481     OUTPUT:
482     neighbors_df - (pandas dataframe) a dataframe
    with:
483                     neighbor_id - is a neighbor
    user_id
484                     similarity - measure of the
    similarity of each user to the provided user_id
485                     num_interactions - the number
    of articles viewed by the user
486 
487     Other Details - sort the neighbors_df by the
    similarity and then by number of interactions
    where
488                     highest of each is higher in
    the dataframe, i.e. Descending order
489 
490     """
491     user_ids, sim = find_similar_users(user_id,
    include_similarity=True)
492 
493     neighbors_df = pd.DataFrame({'neighbor_id':
    user_ids,
494                                  'similarity':
    sim})
495     num_interactions = []
496     for row in range(len(neighbors_df)):
497         id = int(neighbors_df.iloc[row]['
    neighbor_id'])-1
498         num_interactions.append(user_item.iloc[id
    ].sum())
499 
```

```
500     neighbors_df["num_interactions"] =
    num_interactions
501
502     neighbors_df.sort_values(by=["similarity",'
    num_interactions'], ascending=False, inplace=True
    , ignore_index=True)
503
504     return neighbors_df # Return the dataframe
    specified in the doc_string
505 #%%
506 def user_user_recs_part2(user_id, m=10):
507     """
508     INPUT:
509     user_id - (int) a user id
510     m - (int) the number of recommendations you
    want for the user
511
512     OUTPUT:
513     recs - (list) a list of recommendations for
    the user by article id
514     rec_names - (list) a list of recommendations
    for the user by article title
515
516     Description:
517     Loops through the users based on closeness to
    the input user_id
518     For each user - finds articles the user hasn't
     seen before and provides them as recs
519     Does this until m recommendations are found
520
521     Notes:
522     * Choose the users that have the most total
    article interactions
523     before choosing those with fewer article
    interactions.
524
525     * Choose articles with the articles with the
    most total interactions
526     before choosing those with fewer total
    interactions.
527
```

```
528     """
529     top_sorted_users = get_top_sorted_users(
    user_id)
530     recs = [] # initalize recs
531     user = user_item.iloc[user_id] # initialize
    user
532
533     for u in range(0,len(top_sorted_users)):
534         id = top_sorted_users["neighbor_id"][u]
535         sim_user = user_item.iloc[id]
536         result = user_item.columns[(sim_user == 1
    ) & (user == 0)].tolist()
537
538         recs.extend([item for item in result if
    item not in recs])
539
540         # Check if we have enough recommendations
541         n_recs = len(recs)
542         if n_recs >= m:
543             break  # Exit loop if we have enough
    recommendations
544
545     recs = recs[:m]
546     recs = list(map(int, recs))
547
548     # sort articles by most total interaction
549     recs_df = df[df["article_id"].isin(recs)]
550     recs_df = recs_df.groupby("article_id")["
    user_id"].count().reset_index()
551     recs_df.columns = ["article_id", "count"]
552     recs_df = recs_df.sort_values(by="count",
    ascending=False).reset_index(drop=True)
553
554     recs = recs_df["article_id"].tolist()
555
556     return recs, get_article_names(recs) # return
    your recommendations for this user_id
557 #%%
558 # Quick spot check - don't change this code - just
     use it to test your functions
559 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)
#%% md
`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.
#%%
print(get_top_sorted_users(1, user_item=user_item).head(n=1))
print(get_top_sorted_users(2, user_item=user_item).head(n=10))
print(get_top_sorted_users(131, user_item=user_item).head(n=10))
#%%
### Tests with a dictionary of results
user1_most_sim =  get_top_sorted_users(1)["neighbor_id"][0]# Find the user that is most similar to user 1
user2_6th_sim =  get_top_sorted_users(2)["neighbor_id"][5]# Find the 6th most similar user to user 2
user131_10th_sim =  get_top_sorted_users(131)["neighbor_id"][9]# Find the 10th most similar user to user 131
#%%
## Dictionary Test Here
sol_5_dict = {
    'The user that is most similar to user 1.': user1_most_sim,
    'The user that is the 6th most similar to user 2.': user2_6th_sim,
    'The user that is the 10th most similar to user 131.': user131_10th_sim,
}
```

```
583
584 t.sol_5_test(sol_5_dict)
585 #%% md
586 `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.
587 #%% md
588 Answer:
589
590 If no user history is available, we can not use a
    recommendation based on user similarity. Therefore
    , we could only recommend articles based on how
    many interactions they have: *get_top_articles*
591 #%% md
592 `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.
593 #%%
594 # What would your recommendations be for this new
    user 0?  As a new user, they have no observed
    articles.
595 # Provide a list of the top 10 article ids you
    would give to
596 new_user_recs = get_top_article_ids(10)
597
598 #%%
599 assert set(new_user_recs) == {1314, 1429, 1293,
    1427, 1162, 1364, 1304, 1170, 1431, 1330}, "Oops
    !  It makes sense that in this case we would want
    to recommend the most popular articles, because we
     don't know anything about these users."
600
601 print("That's right!  Nice job!")
602 #%% md
603 ### <a class="anchor" id="Content-Recs">Part IV:
```

```
603 Content Based Recommendations</a>
604
605 Another method we might use to make
    recommendations is to recommend similar articles
    that are possibly related. One way we can find
    article relationships is by clustering text about
    those articles.  Let's consider content to be the
    article **title**, as it is the only text we have
    available. One point to highlight, there isn't one
     way to create a content based recommendation,
    especially considering that text information can
    be processed in many ways.
606
607 `1.` Use the function bodies below to create a
    content based recommender function `
    make_content_recs`. We'll use TF-IDF to create a
    matrix based off article titles, and use this
    matrix to create clusters of related articles. You
     can use this function to make recommendations of
    new articles.
608 #%%
609 df.head()
610 #%%
611 from sklearn.cluster import KMeans
612 from sklearn.feature_extraction.text import
    TfidfVectorizer
613 from sklearn.pipeline import make_pipeline
614 from sklearn.preprocessing import Normalizer
615 from sklearn.decomposition import TruncatedSVD
616 #%%
617 # unique articles
618 df_unique_articles = df.drop_duplicates(subset=["
    article_id"], keep="first")
619 #%%
620 # Create a vectorizer using TfidfVectorizer and
    fit it to the article titles
621 max_features = 200
622 max_df = 0.75
623 min_df = 5
624
625 vectorizer = TfidfVectorizer(
```

```
626     max_df=max_df,
627     min_df=min_df,
628     stop_words="english",
629     max_features=max_features,
630 )
631 print("Running TF-IDF")
632 X_tfidf = vectorizer.fit_transform(
    df_unique_articles["title"]) # Fit the vectorizer
    to the article titles
633
634 print(f"n_samples: {X_tfidf.shape[0]}, n_features
    : {X_tfidf.shape[1]}")
635
636 lsa = make_pipeline(TruncatedSVD(n_components=50
    ), Normalizer(copy=False))
637 X_lsa = lsa.fit_transform(X_tfidf)# Fit the LSA
    model to the vectorized article titles
638 explained_variance = lsa[0].
    explained_variance_ratio_.sum()
639
640 print(f"Explained variance of the SVD step: {
    explained_variance * 100:.1f}%")
641 #%%
642 # Let's map the inertia for different number of
    clusters to find the optimal number of clusters
643 # We'll plot it to see the elbow
644 inertia = []
645 clusters = 300
646 step = 25
647 max_iter = 50
648 n_init = 5
649 random_state = 42
650 for k in range(1, clusters, step):
651     kmeans = KMeans(
652         n_clusters=k,
653         max_iter=max_iter,
654         n_init=n_init,
655         random_state=random_state,
656     ).fit(X_lsa)
657     # inertia is the sum of squared distances to
    the closest cluster center
```

```
658     inertia.append(kmeans.inertia_)
659 plt.plot(range(1, clusters, step), inertia)
660 plt.xlabel('Number of clusters')
661 #%% md
662 There appears to be an elbow about 50, so we'll
    use 50 clusters.
663 #%%
664 n_clusters = 50
665 kmeans = KMeans(
666         n_clusters=n_clusters,
667         max_iter=max_iter,
668         n_init=n_init,
669         random_state=random_state,
670 ).fit(X_lsa)
671 #%%
672 # create a new column `title_cluster` and assign
    it the kmeans cluster labels
673 # First we need to map the labels to
    df_unique_articles article ids and then apply
    those to df
674
675 pred_cluster = kmeans.predict(X_lsa)
676 article_cluster_map = dict(zip(df_unique_articles[
    'article_id'], pred_cluster))
677
678 df['title_cluster'] = df['article_id'].map(
    article_cluster_map)# apply map to create title
    clusters
679 #%%
680 # Let's check the number of articles in each
    cluster
681 np.array(np.unique(kmeans.labels_, return_counts=
    True)).T
682 #%%
683 def get_similar_articles(article_id, df=df):
684     """
685     INPUT:
686     article_id - (int) an article id
687     df - (pandas dataframe) df as defined at the
    top of the notebook
688
```

```
689     OUTPUT:
690     article_ids - (list) a list of article ids
    that are in the same title cluster
691
692     Description:
693     Returns a list of the article ids that are in
    the same title cluster
694     """
695      # Your code here
696     title_cluster = df.loc[df["article_id"] ==
    article_id, "title_cluster"].values[0]
697     articles_in_cluster = df[df["title_cluster"]==
    title_cluster]
698
699     # remove the input article_id from the list
700     articles_in_cluster = articles_in_cluster[
    articles_in_cluster["article_id"] != article_id]
701
702     articles_in_cluster = articles_in_cluster["
    article_id"].unique().tolist()
703
704     return articles_in_cluster
705 #%%
706 def make_content_recs(article_id, n, df=df):
707     """
708     INPUT:
709     article_id - (int) an article id
710     n - (int) the number of recommendations you
    want similar to the article id
711     df - (pandas dataframe) df as defined at the
    top of the notebook
712
713     OUTPUT:
714     n_ranked_similar_articles - (list) a list of
    article ids that are in the same title cluster
    ranked
715                                     by popularity
716     n_ranked_article_names - (list) a list of
    article names associated with the list of article
    ids
717
```

```
    Description:
    Returns a list of the n most ranked similar articles to a given article_id based on the title cluster in df. Rank similar articles using the function get_ranked_article_unique_counts.
    """
    similar_articles = get_similar_articles(article_id)

    n_ranked_similar_articles = get_ranked_article_unique_counts(similar_articles)[:n]
    n_ranked_similar_articles = [sublist[0] for sublist in n_ranked_similar_articles]

    n_ranked_article_names = get_article_names(n_ranked_similar_articles)

    return n_ranked_similar_articles, n_ranked_article_names

#%%
# Test out your content recommendations given artice_id 25
rec_article_ids, rec_article_titles = make_content_recs(25, 10)
print(rec_article_ids)
print(rec_article_titles)
#%%
assert len({1025, 593, 349, 821, 464, 29, 1042, 693, 524, 352}.intersection(set(rec_article_ids))) > 0, "Oops! Your the make_content_recs function doesn't work quite how we expect."
#%% md
`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? What other text data would be useful to help make better recommendations besides the article title?
```

```
740 #%% md
741 **Answer:**
742
743 The content-based recommendation system works by
    extracting information from article titles using
    natural language processing (TF-IDF). A vocabulary
     of 125 terms is extracted and used to cluster the
     titles into 50 distinct clusters. The number of
    clusters was determined by analyzing the elbow
    point of the scree plot.
744
745 When recommending articles, those that belong to
    the same cluster as the provided article are
    suggested, with higher-ranked articles being
    prioritized.
746
747 To improve the content-based recommendation system
    , it would be beneficial to include additional
    information beyond just the title. Specifically,
    the article text data would provide deeper
    insights into the full article content.
    Additionally, metadata such as the publication
    date and author could further enhance the
    clustering process, by including factors like
    recency and authorship, allowing the system to
    make more relevant and timely recommendations.
748
749 #%% md
750 ### <a class="anchor" id="Matrix-Fact">Part V:
    Matrix Factorization</a>
751
752 In this part of the notebook, you will build use
    matrix factorization to make article
    recommendations to users.
753
754 `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.
755 #%%
```

```
# quick look at the matrix
user_item.head()
#%% md
`2.` In this situation, you can use Singular Value Decomposition from [scikit-learn](https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html) on the user-item matrix.  Use the cell to perform SVD.
#%%
from sklearn.decomposition import TruncatedSVD
from sklearn.metrics import precision_score, recall_score, accuracy_score
# Using the full number of components which equals the number of columns
svd = TruncatedSVD(n_components=len(user_item.columns), n_iter=5, random_state=42)

u = svd.fit_transform(user_item)
v = svd.components_
s = svd.singular_values_
print('u', u.shape)
print('s', s.shape)
print('vt', v.shape)
#%% md
`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 better metrics when making predictions for the 1 and 0 values in the user-item matrix.  Run the cell below to get an idea of how our metrics improve as we increase the number of latent features.
#%% md

#%%
num_latent_feats = np.arange(10, 700+10, 20)
metric_scores = []

for k in num_latent_feats:
    # restructure with k latent features
    u_new, vt_new = u[:, :k], v[:k, :]
```

```
783
784     # take dot product
785     user_item_est = abs(np.around(np.dot(u_new,
    vt_new))).astype(int)
786     # make sure the values are between 0 and 1
787     user_item_est = np.clip(user_item_est, 0, 1)
788
789     # total errors and keep track of them
790     acc = accuracy_score(user_item.values.flatten
    (), user_item_est.flatten())
791     precision = precision_score(user_item.values.
    flatten(), user_item_est.flatten())
792     recall = recall_score(user_item.values.flatten
    (), user_item_est.flatten())
793     metric_scores.append([acc, precision, recall])
794
795
796 plt.plot(num_latent_feats, metric_scores, label=['
    Accuracy', 'Precision', 'Recall'])
797 plt.legend()
798 plt.xlabel('Number of Latent Features')
799 plt.title('Metrics vs. Number of Latent Features')
800 #%% md
801 `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. Given
    the plot above, what would you pick for the number
     of latent features and why?
802 #%% md
803 **Answer:**
804
805 I would select the number of latent features at
    the elbow point which occurs around 200 latent
    features. While increasing the number of latent
    features improves recall, it could also make the
    model overly complex. It is important to find a
    balance between computational complexity and model
     performance, especially since the improvements
    stagnate after the elbow point.
```

```
806 #%% md
807
808 #%% md
809 `5.` Using 200 latent features and the values of U
    , S, and V transpose we calculated above, create
    an article id recommendation function that finds
    similar article ids to the one provide.
810
811 Create a list of 10 recommendations that are
    similar to article with id 4.  The function should
     provide these recommendations by finding articles
     that have the most similar latent features as the
     provided article.
812 #%%
813 article_id = 100
814 article_id = str(article_id)
815 article_idx= user_item.columns.tolist().index(
    article_id)
816 vt = v
817 cos_sim = cosine_similarity(vt.T)
818
819 article_sim = pd.DataFrame({"sim": cos_sim[
    article_idx],
820                             "article_id":
    user_item.columns.values
821                             })
822 article_sim = article_sim.drop(article_idx)
823
824 article_sim.sort_values(by="sim", ascending=False
    , inplace=True, ignore_index=True)
825
826 similar_article_id = article_sim["article_id"]
827 similarity = article_sim["sim"]
828
829 #%%
830 def get_svd_similar_article_ids(article_id, vt,
    user_item=user_item, include_similarity=False):
831     """
832     INPUT:
833     article_id - (int) an article id
834     vt - (numpy array) vt matrix from SVD
```

```
835     user_item - (pandas dataframe) matrix of users
     by articles:
836                 1's when a user has interacted
    with an article, 0 otherwise
837     include_similarity - (bool) whether to include
     the similarity in the output
838
839     OUTPUT:
840     article_ids - (list) a list of article ids
    that are in the same title cluster
841
842     Description:
843     Returns a list of the article ids similar
    using SVD factorization
844     """
845     # Find the index of the article_id
846     article_id = str(article_id)
847     article_idx= user_item.columns.tolist().index(
    article_id)
848     # Find the cosine similarity of all articles
849     cos_sim = cosine_similarity(vt.T)
850     # Get similarities only for the cos_sim of the
     article_idx
851     article_sim = pd.DataFrame({"sim": cos_sim[
    article_idx],
852                                 "article_id":
    user_item.columns.values
853                                 })
854
855     article_sim = article_sim.drop(article_idx)
856     # Sort and return the articles, don't include
    the own article
857     article_sim.sort_values(by="sim", ascending=
    False, inplace=True, ignore_index=True)
858
859     similar_article_id = article_sim["article_id"
    ].astype(int)
860     similarity = article_sim["sim"]
861
862     if include_similarity:
863         return [[similar_article_id, similarity]]
```

```
    return similar_article_id.tolist()
#%%
# Create a vt_new matrix with 200 latent features
k = 200
vt_new = v[:k, :]
#%%
# What is the article name for article_id 4?
print("Current article:", get_article_names([4], df=df)[0])
#%%
# What are the top 10 most similar articles to article_id 4?
rec_articles = get_svd_similar_article_ids(4, vt_new, user_item=user_item)[:10]
rec_articles
#%%
# What are the top 10 most similar articles to article_id 4?
get_article_names(rec_articles, df=df)
#%%
assert set(rec_articles) == {1199, 1068, 486, 1202, 176, 1120, 244, 793, 58, 132}, "Oops! Your the get_svd_similar_article_ids function doesn't work quite how we expect."
print("That's right!  Great job!")
#%%
make_content_recs(article_id=4, n=10)
#%% md
`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 above are an improvement to how users currently find articles, either by Sections 2, 3, or 4? Add any tradeoffs between each of the methods, and how you could leverage each type for different situations including new users with no history, recently new users with little history, and users with a lot of history.
#%% md
**Answer:**
```

Each recommendation model has its strengths and limitations. While rank-based models do not provide personalized recommendations, they can work well for new users without prior history. For example, they are useful for a homepage showcasing the most frequently interacted-with articles. However, this approach tends to favor popular articles, making it less effective for newer or niche articles.

User-user based collaborative filtering provides personalized recommendations based on similar users' preferences. This makes it ideal for sections that suggest articles liked by others with similar interests. However, since it does not consider article content, recommendations may not always be thematically relevant. Additionally, this method required user history and is computationally intensive due to pairwise similarity calculations.

In contrast, content-based recommendations suggest articles based on their content. This approach is useful for recommending articles closely related to the one a user is currently reading. It also works well for new users, as it does not rely on user history. However, it may limit diversity, as it tends to recommend articles within the same topic range.

Matrix-factorization provide recommendations based on hidden latent factors. This model is effective with sparse data and can, therefore, be used in situations where only little user history is available. However, this model lacks interpretability, making it difficult to explain why a particular article was recommended and what the hidden latent factors represent.

To evaluate the effectivenss of these models, A/B

testing can be implemented. Key metrics such as conversion rate, click-trough rate, and user engagement can help determine performance. Additionally, interleaved testing (presenting two recommendation models simultaneously) can be used to directly compare two models and evaluate which recommendations where preferred.

Finally, since each model produces different recommendations, combining them may improve overall performance. For example, content-based recommendations for *article_id 4* focus primarily on the analytics aspect of the article title, while matrix factorization suggests a more diverse set of articles. A hybrid approach, blending content-based filtering for relevance with matrix factorization for diversity, could enhance recommendation quality by balancing specificity with broader exploration. The hybrid approach would of course need to be properly tested.

```
#%%
#from subprocess import call
#call(['python', '-m', 'nbconvert', '--to html', '
Recommendations_with_IBM.ipynb'])
```